TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Chair of Computer Networks

On-Demand Composition of Smart Service Systems in Decentralized Environments

The RoleDiSCo Approach

Markus Wutzler, M.Sc.

Dissertation
submitted to
Technische Universität Dresden, Faculty of Computer Science
in partial fulfillment of the requirements in order to obtain the academic degree
Doktor-Ingenieur (Dr.-Ing.)

Dresden, June 2018

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Chair of Computer Networks

On-Demand Composition of Smart Service Systems in Decentralized Environments

The RoleDiSCo Approach

Markus Wutzler, M.Sc.
born on January 3, 1991 in Freital, Germany

Dissertation
submitted to
Technische Universität Dresden, Faculty of Computer Science
in partial fulfillment of the requirements in order to obtain the academic degree
Doktor-Ingenieur (Dr.-Ing.)

Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill *first reviewer*
Technische Universität Dresden, Germany

Prof. Dr. rer. nat. Markus Endler *second reviewer*
Pontifícia Universidade Católica do Rio de Janeiro, Brazil

Prof. Dr.-Ing. Thomas Schlegel *advisor*
Hochschule Karlsruhe – Technik und Wirtschaft, Germany

Date of Submission
November 17, 2017

Date of Defense
April 4, 2018

Für meinen Opa.

To my Grandpa.

The most profound technologies are those that disappear.
They weave themselves into the fabric of everyday life
until they are indistinguishable from it.

— *Mark Weiser*

Abstract

The increasing number of smart systems inevitably leads to a huge number of systems that potentially provide independently designed, autonomously operating services. In near-future smart computing systems, such as smart cities, smart grids or smart mobility, independently developed and heterogeneous services need to be dynamically interconnected in order to develop their full potential in a rather complex collaboration with others. Since the services are developed independently, it is challenging to integrate them on-the-fly at run time. Due to the increasing degree of distribution, such systems operate in a decentralized and volatile environment, where central management is infeasible. Conversely, the increasing computational power of such systems also supersedes the need for central management. The four identified key problems of adaptable, collaborative *Smart Service Systems* are on-demand composition of complex service structures in decentralized environments, the absence of a comprehensive, serendipity-aware specification, a discontinuity from design-time specification to run-time execution, and the lack of a development methodology that separates the development of a service from that of its role essential to a collaboration.

This approach utilizes role-based models, which have a collaborative nature, for automated, on-demand service composition. A rigorous two-phase development methodology is proposed in order to demarcate the development of the services from that of their role essential to a collaboration. Therein, a collaboration designer specifies the collaboration including its abstract functionality using the proposed role-based collaboration specification for Smart Service Systems. Thereof, a partial implementation is derived, which is complemented by services developed in the second phase. The proposed middleware architecture provides run-time support and bridges the gap between design and run time. It implements a protocol for coordinated, role-based composition and adaptation of Smart Service Systems. The approach is quantitatively and qualitatively evaluated by means of a case study and a performance evaluation in order to identify limitations of complex service structures and the trade-off of employing the concept of roles for composition and adaptation of Smart Service Systems.

Acknowledgements

Though receiving a doctor's degree has been a dream since many years even before I started my studies in computer science, I never imagined that I would get the opportunity to realize that dream. In this regard, I particularly want to express my profound gratitude to my supervisor Prof. Dr. Alexander Schill for regularly providing feedback and advice as well as always keeping his eye on the schedule and encouraging me to meet the deadlines. Likewise, I want to express my thanks to the German Research Foundation (DFG) for funding the RoSI Research Training Group, in which I was involved during the past three years. It was a great pleasure to work within this large group and to solely focus on my PhD work. In that sense I also want to thank all the members of the RoSI Research Training group for many interesting and lively discussions, the valuable feedback to my work, and of course the pleasant work atmosphere.

Moreover, I want to express my profound gratitude to Martin Weißbach for not only being a colleague over the past eight years, but also for being a friend and a teammate one can always rely on; to Thomas Springer for acting as a mentor in the day-to-day academic life; and to Nguonly Taing for enriching my mind with perspectives from a different country and culture, thereby, enlightening us to esteem the situation and society we are living in. Thanks for all the valuable, sometimes controversial but eventually always constructive discussions and criticism, the feedback, support and the collective efforts enhancing my papers and presentations.

Additionally, I want to thank the members of the Chair of Computer Networks, especially Marius Feldmann for initially teaching me how to write scientific theses, Christin Groba for discussing the academic aspects of my thesis, as well as Iris Braun, Tenshi Hara and Philipp Grubitzsch for the encouraging conversations about the difficulties and challenges of doing a PhD.

I also want to express my great gratitude to my friends and family. Ivonne Wittig, who proofread my thesis entirely in terms of grammar and orthography, and Sarah Hiller greatly supported me throughout my entire studies. They were always interested in what I am doing and even if they could rarely contribute to the topic of my work, we

also had a lot of valuable discussions. Their moral support was what kept me going on and finishing this thesis. Of course, I want to thank my parents Ines and Mario as well as my grandfather's life partner Brigitte for the constant interest in my work and their great moral support, too.

Eventually, I want to express my most profound gratitude to my grandfather Dieter Prengel himself for supporting me throughout my entire life and believing in whatever I am doing. Though he neither expected me to study nor to do a PhD afterwards, it was his engineer's mind which impressed me and had a lasting effect on me and eventually led me to become who I am today.

Thank you.

Markus Wutzler
Dresden, June 2018

List of Publications

The following peer-reviewed publications cover the main contributions of this thesis:

[MW1] Markus Wutzler. "Composing Adaptive Software Systems in Decentralized Infrastructures". In: *Proceedings of MobiSys 2016 PhD Forum* (Singapore, Singapore). MobiSys Ph.D. Forum '16. New York, NY, USA: ACM, 2016, pp. 15–16. ISBN: 978-1-4503-4331-2. DOI: 10.1145/2930056.2933325.

[MW2] Markus Wutzler. "Exploring On-Demand Composition of Pervasive Collaborations in Smart Computing Environments". In: *On the Move to Meaningful Internet Systems: OTM 2016 Workshops*. Ed. by Ioana Ciuciu, Christophe Debruyne, Herve Panetto, et al. Cham: Springer International Publishing, 2016, pp. 1–10. ISBN: 978-3-319-55961-2. DOI: 10.1007/978-3-319-55961-2_31.

[MW3] Markus Wutzler, Thomas Springer, and Alexander Schill. "Coordinated Composition of Continuous Service Collaborations in Decentralized Smart Computing Environments". In: *Proceedings of the Symposium on Applied Computing*. Symposium on Applied Computing (Pau, France, Apr. 9–13, 2018). SAC'18. Apr. 2018. DOI: 10.1145/3167132.3167145.

[MW4] Markus Wutzler, Thomas Springer, and Alexander Schill. "RoleDiSCo: A Middleware Architecture and Implementation for Coordinated On-Demand Composition of Smart Service Systems in Decentralized Environments". In: *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems*. 2nd Workshop on Engineering Collective Adaptive Systems (eCAS) (Tucson, AZ, USA, Sept. 18–22, 2017). FAS*W. Sept. 2017. DOI: 10.1109/FAS-W.2017.118.

[MW5] Markus Wutzler, Thomas Springer, and Alexander Schill. "Utilizing Role-based Models for On-Demand Composition of Smart Service Systems". In: *Companion to the First International Conference on the Art, Science and Engineering of Programming* (Brussels, Belgium, Apr. 3–6, 2017). Programming '17. New York, NY, USA: ACM, 2017, 13:1–13:6. ISBN: 978-1-4503-4836-2. DOI: 10.1145/3079368.3079390.

[MW6] Markus Wutzler, Martin Weißbach, and Thomas Springer. "Role-Based Models for Building Adaptable Collaborative Smart Service Systems". In: *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*. Second IEEE Workshop on Smart Service Systems (SmartSys 2017) (Hong Kong, China, May 29–31, 2017). IEEE, May 2017, pp. 1–6. ISBN: 978-1-5090-6517-2. DOI: 10.1109/SMARTCOMP.2017.7947041.

The following peer-reviewed publications are closely related to the content of this thesis, but serve rather as a foundation for or a prospective application of the contributions presented in this thesis, and thus are not contained herein:

[MW7] Nguonly Taing, Markus Wutzler, Thomas Springer, Nicolás Cardozo, and Alexander Schill. "Consistent Unanticipated Adaptation for Context-Dependent Applications". In: *COP'16: Proceedings of the 8th International Workshop on Context-Oriented Programming* (Rome, Italy). New York, NY, USA: ACM Press, 2016, pp. 33–38.

[MW8] Martin Weißbach, Nguonly Taing, Markus Wutzler, Thomas Springer, Alexander Schill, and Siobhán Clarke. "Decentralized Coordination of Dynamic Software Updates in the Internet of Things". In: *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)* (Reston, USA, Dec. 12–14, 2016). Dec. 2016, pp. 171–176. DOI: 10.1109/WF-IoT.2016.7845450.

Contents

Acronyms

SASS	Self-Adaptive Software System
SOSS	Self-Organizing Software System
SOA	Service-Oriented Architecture
EBCS	Ensemble-based Component System
NLP	Network Level Protocol
UPnP	Universal Plug 'n' Play
SSDP	Simple Service Discovery
IoT	Internet of Things
SLA	Service Level Agreement
PCC	Pervasive Collaboration Coordinator
DSL	Domain-Specific Language
CROM	Compartment-Role-Object Model
WS-CDL	Web Services Choreography Description Language
BPEL	Business Process Execution Language
BPMN	Business Process Modeling Notation
YAWL	Yet Another Workflow Language
WSDL	Web Services Description Language

The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.

Mark Weiser, [81]

1 Introduction

In 1991, Mark Weiser presented his vision of *»The Computer for the 21st Century«* [81], in which small devices, called *tabs* and *pads*, are wirelessly interconnected with each other and other parts of the infrastructure. He envisioned that such devices are integrated seamlessly and users are not aware of their underlying technology and infrastructure anymore. The increasing number of smart devices [27, 31, 32] indicates that within three decades many of his predictions became true. Besides smartphones, physical *things*, such as heaters, fridges, or TV sets, got interconnected through the Internet of Things (IoT), which makes them accessible from other connected devices.

> The real power of the [application] concept comes not from any one of these devices; it emerges from the interaction of all of them. (Mark Weiser, [81])

Systems utilize their functionality in collaboration with other devices. Within home networks, for instance, simple collaborations are established using low-level network protocols to connect a smart phone and a smart TV. Beyond closed infrastructures, such collaborations are established through services. However, in prospective smart computing environments, such as smart home, smart mobility, smart grid or pervasive health, it is almost infeasible to foresee all possible service collaborations. Instead, independently designed services are dynamically interconnected to *Smart Service Systems*, more precisely *smart* systems of smart services, in order to create synergies. Hence, they collaborate on-demand, but the collaboration's structure might change over time because of participants who join and leave.

Collaborations are one foundation of the *Concept of Roles* [9], which already found its way to distributed systems to some extent. According to Boella and Steimann, a *role* encapsulates an abstract functionality that will be utilized in its *collaboration* with other roles. A *collaboration* is defined as "a structure of collaborating roles." [9] The actual performance of the role is delegated to its dynamically bound *player*. This binding, denoted as *fills* relation, can be annotated with requirements for respective players in order to gain flexibility at run time. Thanks to the dynamic *fills* relation, roles are a promising concept to ease composition and reconfiguration, especially at run time.

1.1 Motivation

While in the past the need for collaboration of systems or applications has been enforced along with their distribution, nowadays, the increasing number of smart systems and their services is not a result of traditional reasons for distribution, such as separation of concerns, load-balancing, security, or performance, all of which cause a rather static distribution. The increasing number of smart systems in the domestic sector [27, 31] is due to people replacing devices, consumer electronics, appliances, etc., with smart ones. In other areas, such as in smart cities or smart factories, introducing smart systems might be driven by improved monitoring and automation which lead to cost optimizations. Consequently, those systems come along with independently developed services, which eventually desire a utilization within a collaboration with other services.

Although more and more devices become smart, their ability to collaborate remains rather limited. At current state of practice, collaborations between smart systems are often vendor-specific, which means that only devices of the same manufacturer are able to collaborate, or collaborations are restricted to simple relationships, *e.g.,* low-level network protocols connecting a smart phone to a smart TV to stream some media.

Future smart computing environments, such as smart homes, smart mobility, smart grid or pervasive health, will heavily rely on the spontaneous collaboration of independently designed services which are dynamically interconnected to *Smart Service Systems* at run time. Though the systems and their services are intended to work independently, their participation within a collaboration of many such services will result in a synergy of the single services and, thus, yields their overall utilization to more than the only sum of the individual parts. Gartner expects that a single family home contains more than 500 smart systems by 2022 [32] and presents the *Connected Home* as one of the current key technology trends [33]. In order to realize such an *intelligent* smart home environment, all these more than 500 systems must be highly interconnected in order to develop the systems' full potential. In the domain of smart mobility, *car-to-x systems* [38], in which connected cars communicate with various entities in their environment in order to detect obstacles or to safely pass roadworks, also require continuous, context-aware collaborations of systems and mechanisms to dynamically interconnect their parts at run time. Consequently, the increasing number of smart systems holds a high potential for complex interactions among them, which has not been fully explored yet.

A single smart home, however, is only a very small subset of the IoT. Considering the number of smart homes comprised in a smart city, it is in evidence that fully

centralized management becomes infeasible for such large-scale operating environments. Conversely, the systems' increasing computing power supersedes the need for central management, as for instance in edge-centric [30] or fog computing [77], the computing load is shifted from the core of the network to its edge, *e.g.*, computing on devices and systems close to the user. This eventually leads to a distributed system in a decentralized environment, in which no statically predefined central coordinator is available.

Apart from spontaneous collaborations, future smart computing environments are faced with contextual changes in both their computational environment and the user's situation. Such challenges are tackled by Self-Adaptive Software Systems (SASSs) [70], more precisely by decentralized SASSs, which are an essential field of research [83]. SASSs with decentralized control, however, are still underrepresented [53].

1.2 Terminology

A first definition of the term *Smart Service Systems* was provided by Barile and Polese [6] in 2010. Therein, smart service systems are defined "as service systems designed for a wise and interacting management of their assets and goals". Such systems are based on interaction (among services) and are able to reconfigure themselves. Furthermore, they have to respect the users' situation as well as the computational environment, which means that they have to be context-aware. The definition's background is a rather economical one, which is why certain technological aspects are missing. It is, however, quite interesting to see that – from the users' perspective – this definition perfectly matches the motivation presented precedingly. This thesis picks up the definition by Barile and Polese and specifies additional, engineering-related aspects.

In this thesis, *Smart Service Systems* are considered a class of systems in which independently developed and autonomously operating services are interconnected on-the-fly in order to develop their full potential. In contrast to common approaches, such as Service-Oriented Architectures (SOAs), composition within Smart Service Systems happens spontaneously, for instance, because of an event triggered in response to changes in the user's situation or the system's computational environment, or in response to a user's action, similar to Self-Organizing Software Systems (SOSSs). Such a composition results either in a goal-oriented service collaboration, which has a predefined workflow and is completed as soon as the goal is reached, such as calling an ambulance in case of emergency [16], or in a continuous service collaboration, such as a lecture in terms of a tech-enhanced classroom environment. In contrast to SOSSs, Smart Service Systems

form a *complex*, and *continuous service collaboration*, which is defined below. Apart from that, a key feature of Smart Service Systems is *serendipity* [15, 16], which is the ability to integrate unforeseen resources, such as new systems, new functionality, or new services, at run time. Another crucial feature is context-awareness, *i.e.,* to cope with changes in the computational environment or the user's situation.

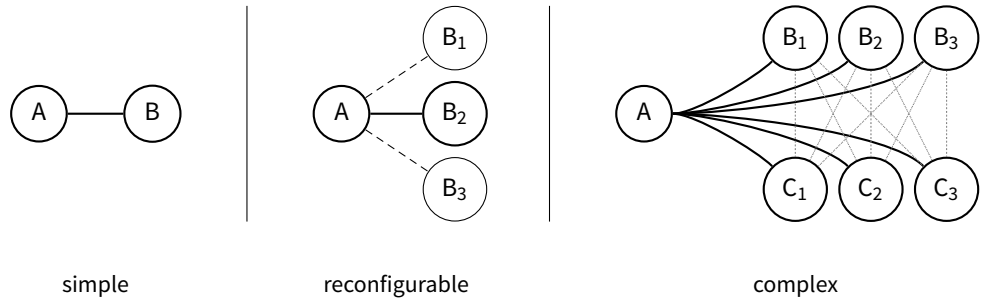


Figure 1.1: Simple, Reconfigurable, and Complex Service Structures.

The *complex* service collaboration refers to the structure of the composition. Figure 1.1 illustrates different types of service structures. Simple service structures are considered to have rather static, predefined bindings between services. Reconfigurable service structures allow for run-time reconfiguration of bindings, *e.g.,* to adapt themselves in response to changing SLAs, but still are unable to bind multiple, similar service instances at the same time. A *complex* service structure additionally can incorporate multiple services with similar *capabilities* simultaneously, *i.e.,* a service can bind multiple instances of another type simultaneously, which is challenging in a decentralized environment. Hence, it is assumed to support relationships other than simply one-to-one as well as multiplicities, constraints and meshed relationships, *i.e.,* services are mutually interconnected. A *capability* denotes an abstract functionality, such as *displaying pictures*, without further specifying its concrete performance.

In contrast to SOSSs, Smart Service Systems do not have a specific goal that can be described by a sequential order of steps in order to be achieved, which means that no predefined control flow can be assumed. Instead, the collaboration is considered *continuous*, or ongoing, until it is terminated explicitly. Thus, communication and interaction between services are asynchronous and bidirectional. The term *service collaboration* emphasizes that services are aware of each other and actively collaborate within a complex service structure independent of an external mediator.

Finally, a *Pervasive Collaboration* denotes a complex, continuous service collaboration at run time, and is defined as a dynamic set of distributed, loosely coupled, on-demand collaborating systems whose main goal is to combine the capabilities of the individual

subsystems in order to create a complex collective system. Pervasive Collaborations are adaptable in the sense of structural reconfiguration and are considered to operate in environments, where no statically predefined central coordinator can be assumed. In other words, a *pervasive collaboration* is an instantiated and running *complex, continuous service collaboration*, which in turn comprises a *complex service structure*, and is considered one approach realizing *Smart Service Systems*. Within a Smart Service System, multiple pervasive collaborations may operate simultaneously and autonomous services may participate in several of them. More specifically, an autonomous service may also participate in the very same pervasive collaboration several times. Finally, an autonomous service may be a Smart Service System itself, which enables to realize systems-of-systems on-demand.

1.3 Problem Statement

Future smart computing environments will heavily rely on the spontaneous collaboration of independently designed and autonomously operating services in order to create synergies and to develop their full potential. How such complex service collaborations can be composed on-the-fly at run time in decentralized environments, where no statically predefined central coordinator is available, is a question yet to be answered.

Smart Service Systems, operating in computing environments such as those mentioned earlier, face three major challenges, which cannot be ›solved‹ but have to be considered throughout this work: *heterogeneity* – the services to be interconnected on-the-fly to form complex service collaborations providing the desired functionalities are independently developed and operate autonomously; *context-awareness* – such a service collaboration will have to adapt to changes in the user's situation or its surrounding computational environment; and *decentralization* – all the aforementioned operating environments are coined by a high degree of distribution and decentralization, thus, statically centralized control of the service composition and subsequent adaptation is infeasible.

Hereinafter, the four key problems of engineering, *i.e.,* developing and running, Smart Service Systems are discussed. These comprise a missing specification for complex, context-aware, continuous service collaborations, the intertwined development processes of the service and its role essential to a collaboration, a discontinuity from design-time specification to run-time composition and adaptation, and eventually the on-demand composition of the desired systems in decentralized environments. For the sake of a better understanding the last two problems are discussed in reverse order as the

discontinuity becomes evident after discussing the design-time specification and the run-time composition as well as adaptation individually.

Missing Specification for Complex, Context-Aware, Continuous Service Collaborations

A collaborative system requires a holistic specification in order to capture the system's overall structure. Thus, a specification for Smart Service Systems needs to cope with collaborations, but above all with context-awareness and serendipity. Existing approaches to specify service collaborations, such as WS-CDL [79], BPEL [78], BPMN [14] or YAWL [1], focus on the workflow and its individual, ordered process steps. This is not applicable for continuous service collaboration, such as in Smart Service Systems, which require a rather structural than process-oriented system specification. Additionally, neither serendipity nor context-awareness are addressed by any of those specifications.

Haesevoets, Weyns, and Holvoet [41] propose a rigorous approach to model complex, dynamic collaborations of distributed services using a notion of roles. Therefore, business processes are functionally decomposed into smaller units to improve modularity and reusability. Their notion of roles is an abstraction of underlying systems to comply with the business process. However, the decomposed business processes enforce a control flow, which contradicts the *continuity* of a service collaboration as well. Hennicker and Klarl [42] propose a formal foundation for modeling distributed systems using a notion of roles as well. In their approach, roles are teamed up into ensembles, which are "groups of active entities that collaborate to perform a certain task." [42] Players in their approach have no intrinsic behavior by definition. Hence, different players do not perform a role differently. In other words, exchanging a player does not result in a different performance of a role, thus, serendipity cannot be achieved. Bucchiarone, Cicchetti, and Sanctis [11] as well as Birdsey, Szabo, and Falkner [8] propose specifications for collective adaptive systems. While Bucchiarone et al. use a notion of roles similar to that of Hennicker et al., the approach by Birdsey et al. relies on agents. Equally to the aforementioned approaches, systems are completely specified, thereby, prohibiting serendipity. Though collaborations are captured to some extent in both approaches, they lack support for context-awareness.

Problem Existing specifications for service collaborations support rather complex service structures but lack serendipity and context-awareness

Goal A specification for Smart Service Systems must capture the context-dependent and collaborative nature of such and needs to preserve the services' independence by allowing for serendipity.

Intertwined Development Processes of Services and Their Collaborations

An essential feature of Smart Service Systems is that the services to be integrated in a collaboration are both *independently developed* and *autonomously operating*. Although service developers are assumed to be interested in integrating their services into collaborations, their sole focus will be on the autonomous service's development but not on that of the collaboration. Service collaborations, however, are considered to be reusable and to integrate autonomous services providing the required functionality on-demand.

In many cases a system is specified in a way that imposes dependencies and leads to a distributed but rather monolithic system since it is only dynamic within its own boundaries. Decoupling the development processes of the autonomous service and its role essential to the collaboration is a fundamental step in order to address serendipity and heterogeneity. In the past, this was addressed twofold: On the one hand, device models with respective interface implementations, such as in the smart home domain, when two smart devices interact with each other using low-level network protocols, *e.g.,* UPnP [63], are limited to collaborations between two systems. On the other hand, holistic service descriptions, such as WSDL [80] in conjunction with BPEL [78], utilized in business processes, require centralized infrastructures.

Problem Intertwined development processes conflict with desired autonomy of an autonomous service and its role essential to a collaboration.

Goal The development processes of the autonomous service and its role essential to a collaboration must be decoupled in order to achieve serendipity.

On-Demand Composition of Complex, Context-Aware, Continuous Service Collaborations

On-demand composition of service systems with respect to predefined application structures or workflows received a lot of attention in the past, especially in centralized environments. The challenges of Smart Service Systems with respect to on-demand composition of complex, context-aware, and continuous service collaborations are above all the system's decentralization and the complex service structure itself.

The decentralized environment prohibits a statically predefined central management of composition and adaptation. Required management components, *e.g.,* directory or discovery services, cannot be centrally provided. Moreover, fully decentralized decision-making, *i.e.,* to agree on a complete system configuration, is another challenging task in decentralized environments [84] as it requires negotiation among all involved systems.

SOSSs [24] deal with the composition of dependent service systems out of many autonomous services and address reconfiguration, *e.g.,* based on utility functions [16]. Their overall structure emerges from internal constraints or mechanisms and is intrinsic to the system. Collective behavior emerges without central control by matching provided and required service ports based on common naming schemes or ontologies. However, SOSSs lack the design-time specification of complex, context-aware service structures. While SOSSs address the challenges of decentralized environments, their composition mechanisms have certain limitations at run time, as it is, for instance, not possible to bind a service to multiple instances of another service type simultaneously. Thus, compositions mostly result in a chain of services, which perfectly matches the goal-oriented nature of SOSSs, but is unsuitable for complex, continuous service collaborations.

Problem Current composition techniques for SOSSs solely rely on simple dependencies, thus, do not allow for composition of complex service collaborations.

Goal A protocol for coordinated composition of complex service collaborations is required which ideally avoids a fully decentralized decision-making process.

Discontinuity from Specification to Composition and Adaptation

The specification of rather complex service structures as well as their composition and adaptation have been approached in isolation which implies certain limitations, such as those described for SOSSs. Keznikl et al. [49] tackled this gap and proposed an Ensemble-based Component System (EBCS), called *DEECo*. The definition of ensembles confers to that of Hennicker and Klarl [42]. *DEECo* addresses both service composition and reconfiguration based on autonomous components. These solely operate on a local knowledge base, which is shared using a common middleware. Hence, there is no active interaction, *i.e.,* collaboration, between autonomous components. Additionally, the system structure is predefined to be a one-to-many relationship and serendipity cannot be achieved as components need to adopt specific interfaces.

Problem Design-time specification of service systems as well as their run-time composition and adaptation have been approached in isolation, leading to a discontinuity in the application's life cycle.

Goal Bridge the gap between design-time specification and automated run-time composition and adaptation in order to achieve a holistic, continuous engineering and operation approach.

Summary

Evidently, the individual problems have been approached almost in isolation: SOSSs achieve dynamic composition and adaptation but lack the specification of complex service structures at design time. Haesevoets, Weyns, and Holvoet [41] as well as Hennicker and Klarl [42] cover this aspect but lack serendipity and context-awareness and thereby dynamic composition and adaptation as well. Keznikl et al. [49] address this gap, but their services do not actively collaborate, the system's structure is rather predefined and serendipity cannot be achieved, all of which hinders the utilization of the approach in Smart Service Systems. Conclusively, the approach to be developed is required to focus on a specification for complex, context-aware, continuous service collaborations, to decouple the development processes, to solve the discontinuity between design-time specification and run-time composition as well as adaptation, and eventually to achieve on-demand composition of Smart Service Systems.

1.4 Requirements Analysis

The central goal of this thesis is to achieve an automated on-demand composition with subsequent adaptation of Smart Service Systems in decentralized environments. Table 1.1 shows the requirements that are posed to the approach which is to be developed within this thesis and which will comprise a development methodology as well as a specification and run-time support for Smart Service Systems.

RQ 1	**Smart Service Systems**
1.1	Complex Service Structures
1.2	Collaborative Nature
1.3	Serendipity
1.4	Context-Awareness
RQ 2	**Demarcated Development Processes**
RQ 3	**Run-Time Support for Smart Service Systems**
3.1	Automated Discovery
3.2	Automated Composition & Adaptation
3.3	Decentralization & Generic Infrastructure Abstraction
RQ 4	**Coordinated Composition and Adaptation of Smart Service Systems**

Table 1.1: Overview of the Requirements.

RQ 1 addresses the requirements posed by the features of Smart Service Systems, explained in Section 1.2. In order to achieve a *spontaneous collaboration* of *independently developed* services, the development processes of the services must be separated from those of potential collaborations, addressed by *RQ 2*. *RQ 3* aims at solving the discontinuity between design and run time in order to eventually achieve a coordinated on-demand composition and adaptation of Smart Service Systems (*RQ 4*) in decentralized environments. Therefore, deriving discovery information and composition as well as adaptation plans must be automated. The run-time support for Smart Service Systems also needs to tackle the challenges of *decentralization* by providing a generic infrastructure abstraction. *RQ 4*, finally, is above all concerned with the coordinated composition and adaptation of Smart Service Systems, *i.e.*, complex, context-aware, and continuous service collaborations.

1.5 Research Questions and Hypothesis

Based on the initial question, how complex service structures can be composed on-the-fly in decentralized environments, the following research questions are derived:

1. *What is a proper abstraction to specify complex service collaborations in order to realize on-demand composition and subsequent adaptation of Smart Service Systems at run time?*

2. *How can these abstractions be supported throughout the application life cycle (i.e., development and operation phase) in a way which preserves the autonomy of both the autonomous service designer and the Smart Service System designer?*

3. *What are limitations of enforcing complex service structures in decentralized environments?*

Chapter 2 will justify the *Concept of Roles* being an intuitive abstraction that perfectly matches the collaborative nature of Smart Service Systems and thereby a promising concept for easing composition and adaptation of such systems. Hence, the approach is based on the hypothesis that *role-based abstractions allow to specify Smart Service Systems at design time and subsequently enable on-demand composition and adaptation of such systems in decentralized environments at run time*. Thereof, a fourth question can be derived:

4. *What is the trade-off of applying the concept of roles to on-demand composition and adaptation of Smart Service Systems? What are limitations?*

The research questions will be answered comprehensively in Chapter 7.

1.6 Focus and Limitations

The focus of this thesis is on engineering and operation of Smart Service Systems in decentralized environments. This comprises a specification, code generation, and methodologies at design time as well as run-time support for composition and adaptation. Hence, the contributions with respect to the research questions comprise:

A Role-based Specification for Smart Service Systems
Roles and collaborations are an intuitive abstraction that perfectly matches the collaborative nature of Smart Service Systems. The clear separation of roles and their players inherently enables serendipity. Hence, role-based modeling approaches are utilized to specify Smart Service Systems at design time. A role-based collaboration specification allows to describe the Smart Service System as a self-contained collaboration. The roles therein represent the abstract functionality an autonomous service should provide to the collaboration. The autonomous service, in turn, acts as the player of the role and provides the concrete performance. [MW5, MW6]

A Two-Phase Development Methodology for Engineering Smart Service Systems
With respect to *RQ 2*, the RoleDiSCo Development Methodology, a rigorous two-phase development methodology, demarcates the development of the service from that of its role essential to a collaboration: First, a collaboration designer specifies the overall collaboration including its abstract functionality using the role-based collaboration specification. Thereof, a partial implementation is derived, which is later complemented with its concrete performance by several other developers in a second phase. This preserves the autonomous development of the service separated from its role essential to a service collaboration. [MW5]

A Middleware Architecture and Implementation for On-Demand Composition and Subsequent Adaptation of Smart Service Systems in Decentralized Environments
The artifacts resulting from the development methodology are designed to be used by the *RoleDiSCo Middleware*, thereby eliminating the existing discontinuity between design and run time (*RQ 3*), and supporting the *Concept of Roles* throughout the entire application life cycle. Additionally, it provides a decentralized discovery mechanism and the foundations for on-demand composition and subsequent adaptation of Smart Service Systems. The resulting research prototype is used to evaluate the RoleDiSCo approach with respect to the research questions. [MW1, MW2, MW4]

A Protocol for Coordinated Composition of Role-based Smart Service Systems
Though the concept of roles is beneficial for automated composition and adaptation,

it poses additional requirements on composition and adaptation processes. Thus, a protocol for coordinated composition of Smart Service Systems is proposed, which above all addresses the challenges of complex service structures (*RQ 4*). [MW3]

Though these contributions cover many aspects of engineering Smart Service Systems, some aspects are explicitly excluded from the scope of this thesis, such as context reasoning and inference in order to holistically determine context. Additionally, mechanisms for reliably performing distributed adaptations in decentralized environments are assumed to exist. As several pervasive collaborations might operate simultaneously, a competitive situation requiring negotiation is conceivable. Such techniques are not investigated in detail within the scope of this thesis. Finally, autonomous services could be implemented using role-based runtimes. This thesis, however, will not cover any contributions concerning local role-based runtimes as those already exist [43, 60, 76].

1.7 Outline

The remainder of this thesis continues with a comprehensive view on the *Concept of Roles* in Chapter 2, in which the perspective on and the understanding of roles used throughout this thesis is explained as well. Subsequently, related state-of-the-art approaches are subject to further discussion in Chapter 3, including role-based modeling abstractions, role-based run-time systems, and approaches that enable distributed systems to collaborate spontaneously. Seeing the aforementioned problem statements confirmed, Chapter 4 presents in detail the RoleDiSCo approach, which tackles the challenges of Smart Service Systems in decentralized environments. First, Section 4.1 introduces the RoleDiSCo Development Methodology, a two-phase development methodology, which demarcates the development processes and includes a role-based specification for Smart Service Systems. Next, Section 4.2 presents the RoleDiSCo Middleware Architecture in order to achieve run-time support for Smart Service Systems in decentralized environments. Eventually, Section 4.3 explains the protocol for coordinated composition and adaptation of Smart Service Systems. Chapter 5 provides insights into the implementation of two research prototypes, which were implemented in order to evaluate the overall RoleDiSCo approach. Chapter 6 qualitatively and quantitatively evaluates the approach by means of a case study and a performance analysis, both relying on the research prototypes. Finally, Chapter 7 concludes this thesis.

2 The Role Concept in Computer Science

In Chapter 1, the *Concept of Roles* was promoted as an intuitive abstraction for engineering Smart Service Systems as its dynamic *fills* relation should ease composition and reconfiguration. This chapter justifies the intuitiveness of the concept of roles by exploring what a role in computer science is and how it is defined or described in literature. Eventually, the terminology concerning roles, collaborations, and players is defined.

2.1 What is a Role in Computer Science?

In the past, several attempts were made to push the concept of roles into computer science. Roles in this domain were mentioned first by Bachman and Daya [4] in the domain of data modeling to distinguish physical entities, *e.g.,* persons or companies, from their role they may play, *e.g.,* customer and supplier. However, a general definition of roles does not exist and this thesis will not attempt to provide one but show where roles appear(ed) in computer science in the past and today.

```
int age = 42;
```

The above line of code shows a variable declaration and, thus, the arguably smallest kind of a role. The value *42* is simply a series of bits (`101010`), which receives its semantics because of the data type and the name of the variable. `101010`, however, could also be the *amount* of items in a stock, or a *character* as it represents the * (asterisk) in ASCII code.

This can be easily lifted to the level of object-oriented programming and modeling languages. There, roles appear in associations between types to denote the association's

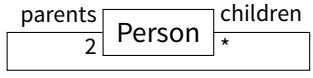


Figure 2.1: Roles as Names of Associations in UML Diagrams.

⟨1⟩	Roles have properties and behaviors.	*(M1, M0)*
⟨2⟩	Roles depend on relationships.	*(M1)*
⟨3⟩	Objects may play different roles simultaneously.	*(M1, M0)*
⟨4⟩	Objects may play the same role (type) several times.	*(M0)*
⟨5⟩	Objects may acquire and abandon roles dynamically.	*(M0)*
⟨6⟩	The sequence of role acquisition and removal may be restricted.	*(M1, M0)*
⟨7⟩	Unrelated objects can play the same role.	*(M1)*
⟨8⟩	Roles can play roles.	*(M1, M0)*
⟨9⟩	Roles can be transferred between objects.	*(M0)*
⟨10⟩	The state of an object can be role-specific.	*(M0)*
⟨11⟩	Features of an object can be role-specific.	*(M1)*
⟨12⟩	Roles restrict access.	*(M0)*
⟨13⟩	Different roles may share structure and behavior.	*(M1)*
⟨14⟩	An object and its roles share identity.	*(M0)*
⟨15⟩	An object and its roles have different identities.	*(M0)*

Table 2.1: Steimann's 15 Classifying Features. (extracted from [73] by Kühn et al. in [58])

end in a relationship [74]. This is exemplified in Figure 2.1, which expresses that a type *Person* has an arbitrary number of *children* and exactly two *parents*. These relationships are realized as attributes, each a collection of *Person* but with different names representing their role. Hence, *Person* has different roles depending on the way an object is accessed. In this notion, roles are simply labels denoting an actually semantic meaning, similar to the notion of roles in role-based access control or multi-agent systems.

Roles in common sense, however, intuitively imply certain behavior that is intrinsic rather to the role than to its actual player, like the role of being a *parent* changes the way of communication compared to the role of being a *supervisor* in a company while still relying on the core communication skills of the *person* acting in this role.

In 2000, Steimann derived a list of 15 classifying features of roles from the then current state of the art of roles in computer science. These features are depicted in Table 2.1 and explained in detail in [73]. It is to be seen as a summary of features found in then available approaches rather than a list of requirements in order to denote something as a role. Kühn et al. [58] annotated these features with their corresponding modeling level, *i.e., M0* referring to instance-level entities, such as objects, and *M1* referring to modeling elements, such as classes or types.

Later, Boella and Steimann [9] covered the behavioral nature of roles in their characterization of roles in the context of computer science: a role encapsulates an abstract functionality which will be utilized in collaboration with other roles. A collaboration is defined as "a structure of collaborating roles, each performing a specialized function, which collectively accomplish some desired functionality. […] [Thus,] a collaboration

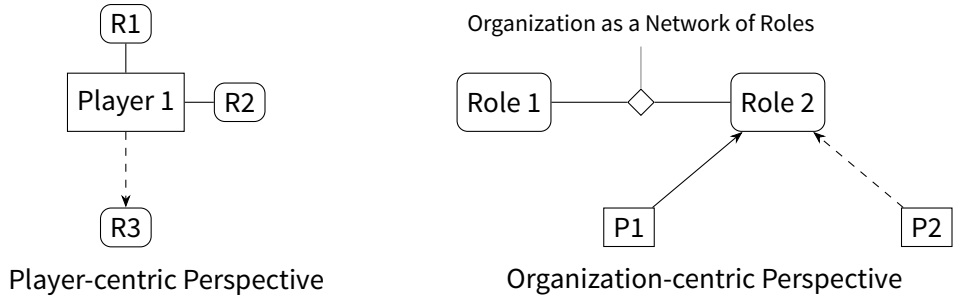


Figure 2.2: Player-centric and Organization-centric Perspectives on Roles. [cf. 20, Figure 1]

structure constitutes a graph where the nodes are roles and the edges are the message interaction paths." [9] The actual performance of the role's abstract functionality is delegated to the role's *player*, which is bound dynamically. This binding is denoted as *plays* or *fills* relation, *i.e.,* a player *plays* or *fills* a role. This *fills* relation may also define requirements, such as class types, required methods or contextual properties, which allows to dynamically select players and bind them to roles at run time.

Colman and Han [20] identified two perspectives on roles, depicted in Figure 2.2: player-centric and organization-centric, respectively. In the player-centric view, which addresses the behavioral nature, roles are attached to a player, which is the stable core object, in order to add, remove or manipulate the player's behavior. Roles are carriers of role-specific or context-dependent state and behavior but cannot exist on their own. The organization-centric view addresses more the collaborative nature of roles. The role's identity and existence depends on the organization defining the role and its associations to other roles but not on the existence of the player. Players complement the roles' abstract behavior and start exposing the behavior when joining in an organization that requires the respective role. The organization-centric perspective of roles follows more the characteristics of roles in human organizations, such as a company. The role itself defines an abstract functionality and the player is responsible for executing this functionality. Colman et al. denote this pattern as *player-as-executor*, which also corresponds to the preceding characterization of roles by Boella and Steimann [9].

Riehle and Gross [69] address the relational nature of roles and propose four types of constraints restricting relationships between roles. For a pair of role types (A, B), *role-dontcare* imposes no restriction at all; *role-implication* $(A \rightarrow B)$ requires an object playing A to play role B as well; *role-equivalence* $(A \leftrightarrow B)$ obliges an object playing A to play role B and vice versa; and *role-prohibition* $(A \dashv B)$ prohibits an object to play roles A and B simultaneously.

⟨16⟩	Relationships between roles can be constrained.	*(M1)*
⟨17⟩	There may be constraints between relationships.	*(M1)*
⟨18⟩	Roles can be grouped and constrained together.	*(M1)*
⟨19⟩	Roles depend on compartments.	*(M1, M0)*
⟨20⟩	Compartments have properties and behaviors.	*(M1, M0)*
⟨21⟩	A role can be part of several compartments.	*(M1, M0)*
⟨22⟩	Compartments may play roles like objects.	*(M1, M0)*
⟨23⟩	Compartments may play roles which are part of themselves.	*(M1, M0)*
⟨24⟩	Compartments can contain other compartments.	*(M1, M0)*
⟨25⟩	Different compartments may share structure and behavior.	*(M1)*
⟨26⟩	Compartments have their own identity.	*(M0)*

Table 2.2: Kühn's Additional Classifying Features of Roles. [58]

Kühn et al. [58] extended Steimann's classification to cover the collaborative (referred to as context-dependent) nature of roles. The additional 11 classifying features, derived from literature, are listed in Table 2.2. Kühn et al. introduced the term *Compartment* as a generalization of different terminologies defining groups of collaborating roles. A *Compartment* denotes "an objectified collaboration with a limited number of participating roles and a fixed scope." [58] Hereinafter, the numbers in ⟨. . .⟩ refer to the respective features listed in Tables 2.1 and 2.2.

2.2 Roles in RoleDiSCo

By definition, the two perspectives on roles are incompatible with each other. Considering, for instance, the interaction, which takes place between players in the player-centric and between roles in the organization-centric perspective it seems impossible to integrate them in one solution. However, applying them on different levels abstraction, both perspectives conceptually complement each other, as illustrated in Figure 2.3. The organization-centric perspective is considered on the level of distributed systems

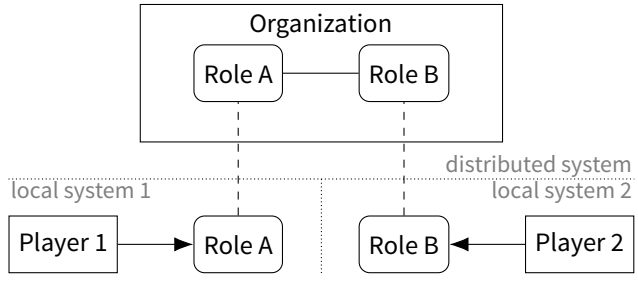


Figure 2.3: Conceptually Complemented Perspective on Roles.

and the player-centric perspective on the level of local runtimes, respectively. Hence, intra-system interaction may take place between players (depending on a concrete local runtime implementation) and inter-system interaction will happen between roles. Complementing the two perspectives allows for a certain degree of heterogeneity as two local runtimes may be different as long as they agree on a common role concept on the level of the organizational perspective.

The perspective adopted in this thesis conforms to the characterization given by Boella and Steimann [9] and follows the organization-centric perspective, defined by Colman and Han [20]. Both consider roles to capture an abstract functionality, the performance of which is delegated to a dynamically bound player. Additionally, both emphasize the collaborative nature and organizational structure of roles. Evidently, the organization-centric perspective on roles perfectly matches the organizational structure of Smart Service Systems, in which the organization maps to the collaboration of autonomous services. Thereby, complex service structures can easily be covered. Roles inherently enable serendipity, since the role and its player are disjoint entities, and different players may perform a role differently. The role is considered the part of the autonomous service essential to its participation within a collaboration. The player, consequently, is the autonomous service itself, at least from the collaboration's perspective. Within the autonomous service, the player may be only a part of the whole service. Hereinafter, the following terminology is used throughout the remainder of this thesis:

Collaboration

Collaborations were already introduced in Section 1.2. In order to streamline those definitions with the terminology of roles, a collaboration is considered a self-contained set of collaborating roles and thereby defines a valid system configuration by means of roles and their relationships, including multiplicities and constraints. It is similar to the objectified collaboration, or *Compartment*, described by Kühn et al. [58]. A role thereby is considered to depend on a specific collaboration ⟨16⟩. With respect to *pervasive collaborations* also the role features ⟨20⟩, ⟨22⟩, and ⟨26⟩ are addressed.

Role

A role encapsulates an abstract functionality, *i.e.,* method interface and partial implementation, and thereby is required to have behavior and state ⟨1⟩. Thus, it provides the interface essential to the collaboration, *i.e.,* the part accessible by other roles within the same collaboration. The *role type* refers to the specification of the role within the specification of the collaboration, whereas the *role instance* refers to the instance bound to a player at run time.

Relationship

Roles are linked through relationships, which thereby define the collaboration's structure and the potential paths of interaction. These relationships are characterized by multiplicities, *i.e.*, one-to-one or one-to-many, and the aforementioned constraints ⟨16⟩. Relationships are important in order to semantically structure the application, *e.g.*, to enforce that two roles are not played by the same player or, with respect to Smart Service Systems, are not provided by the same subsystem within the same collaboration.

Player

A player provides the actual performance of a role by means of complementing the partial implementation of the role's behavior. In order to participate in several pervasive collaborations, the role features ⟨3⟩, ⟨4⟩, and ⟨5⟩ must be addressed by a local role-based runtime, the development of which is not within the scope of this thesis.

Context

Context with respect to literature related to the concept of roles has several meanings. Some approaches consider context a collaboration or a team, while some others consider context a discrete situation. That is why Kühn et al. introduced the notion of *Compartments* [58] in order to capture those various kinds of context. Context with respect to context-awareness of Smart Service Systems refers to the user's situation or the computational environment. Hence, context exists as a separate entity orthogonal to the collaboration and the roles themselves. Thereby it is considered an additional state property of the collaboration or the role reflecting the user's situation or the computational environment. For instance, a collaboration that invites participants based on their current location would require role with a context property providing the location information. A more technical understanding of how context is treated in this thesis is provided in Sections 4.1.1 and 4.2.2.

Adaptive (Sub-)System

A pervasive collaboration is considered an adaptive system because its structure can be recomposed at run time. The subsystem refers to a runtime providing one or more players (and respective role instances) which is adaptive because it could acquire new or drop existing roles. The terms *subsystem* and *node* are used interchangeably throughout this thesis. Subsystem, however, usually refers to an entity part of a pervasive collaboration while node refers to entities in the (network) infrastructure not necessarily part of a pervasive collaboration.

3 State of the Art & Related Work

The previous chapter justified the *Concept of Roles* being an intuitive abstraction for the specification of Smart Service Systems and thereby a promising concept for easing composition and adaptation of such. This chapter analyzes role-based modeling approaches, role-based run-time systems, and approaches for spontaneous collaborations of systems for their utilization to engineer Smart Service Systems.

3.1 Role-based Modeling Abstractions for Software Systems

Steimann [73] and Kühn et al. [58] provided not only the aforementioned lists of in total 26 classifying features of roles but also applied them to role-based modeling and programming languages. As this thesis follows the organization-centric perspective on roles, the extended classification of Kühn et al. is of special interest due to their notion of *Compartments* and serves as a starting point for the subsequent analysis. In order to analyze the current state of practice, a set of classification criteria for role-based modeling abstractions, based on the provided classifying features, is derived from the requirements posed by Smart Service Systems. Subsequently, the current state of practice is analyzed. Conclusively, the findings are summarized and open research gaps as well as remaining challenges are derived.

3.1.1 Classification

The classification criteria for role-based modeling approaches are derived from requirement *RQ 1*, *i.e.*, the engineering aspects of *Smart Service Systems*, including the definition of *complex service structures*, the *collaborative nature*, as well as *serendipity* and *context-awareness*, as discussed in Section 1.4. As the concept of roles perfectly matches the nature of Smart Service Systems, the requirements can be mapped to some of the classifying role features listed in Tables 2.1 and 2.2, to which the numbers in $\langle \ldots \rangle$ refer.

A Smart Service System, first of all, is considered a collaboration of autonomous services. The *collaboration* denotes a self-contained unit of interacting roles, which conforms to the notion of *Compartment*, defined by Kühn et al. [58], and the characterization by Boella and Steimann [9]. As an autonomous service should play the role *essential* to a collaboration in order to participate in that, ⟨19⟩, *i.e.*, roles depend on compartments, of the classification of Kühn et al. can be applied directly. Thus, a role type is known to belong to a specific collaboration type and the collaboration thereby becomes a self-contained entity. More precisely, a collaboration type defines its role types, which in turn are only part of that specific collaboration type. Additionally, the requirement to capture contextual information requires that collaborations have state information ⟨20⟩. The potentially recursive structure, mentioned in Section 1.2, relates to collaborations playing roles like players ⟨22⟩, which in turn implies that collaborations have behavior as well ⟨20⟩, since this is required for players, as stated below.

MA 1 Collaborations

1. Collaborations are self-contained units. ⟨19⟩
2. Collaborations have behavior and state. ⟨20⟩
3. Collaborations may act as players for roles. ⟨22⟩

As mentioned, the role encapsulates the autonomous service's abstract functionality essential to the role's enclosing collaboration and thereby is required to have state and behavior ⟨1⟩. State, here, also captures contextual information. The collaborative nature of Smart Service Systems additionally requires that services interact directly with each other, which is covered by the concept of roles in general since interaction (within the collaboration) happens conceptually between roles. A specific role feature denoting this requirement does not exist. Role constraints, such as those proposed by Riehle and Gross [69], allow to specify complex service structures in a fine-grained way. Hence, constraining relationships between roles ⟨16⟩ is at least a soft requirement in order to weigh approaches which only differ in this particular criterion.

MA 2 Roles

1. Roles have behavior and state. ⟨1⟩
2. Roles define the player's part essential to its participation within a collaboration, hence, depend on collaborations. ⟨19⟩
3. Interaction between roles, not between players. (Collaborative Nature)
4. Relationships between roles can be constrained. ⟨16⟩

Concerning the autonomous service itself, which is considered the player, Steimann [73] refers to it as *Object*. Already at design time, the service's independence needs to be con-

sidered and respected, which is not specifically mentioned by Steimann. The autonomy of the player relates to data and functionality owned by the service, *i.e.,* the player has state and behavior as well. Additionally, in order to achieve variability and flexibility, a role's and player's behavior need to intertwine. Presumably, this was taken for granted, but here it is specifically mentioned as the player's behavior provides autonomy, flexibility and variability to the overall performance of the Smart Service System and thereby addresses serendipity. Roles may define requirements for their players through the *fills* relation, mentioned earlier. In order to achieve variability, players need to be described rather by their shape, *i.e.,* method signatures or contextual properties, than by their type.

Features applying to the modeling level (*M1*) but not explicitly mentioned afore are neither required nor prohibited. Apart from requirements on the level of conceptual modeling elements, a processable specification, such as a Domain-Specific Language (DSL), is required, as it is an essential prerequisite in order to bridge the gap between design-time specification and run-time execution.

MA 3 Players
1. Players have own state and behavior.
2. Behavior of role and player may intertwine.
3. Player's independence can be preserved. (Serendipity, no type restrictions)

MA 4 Processable Specification

Hereinafter, the following symbols indicate the degree of fulfilling a requirement, *i.e.,* ■ denotes complete fulfillment, ⊞ denotes partial fulfillment, and □ denotes no fulfillment.

3.1.2 Approaches

The survey conducted by Kühn et al. [58] serves as a basis for the subsequent state-of-the-art analysis and is extended by other role-based modeling abstractions appeared later or not covered afore. Approaches not following the organizational perspective [20], *i.e.,* not fully supporting ⟨19⟩ with respect to Kühn's classification, are not covered below since MA 1 is the most essential requirement for specifying Smart Service Systems.

With respect to this exclusion criterion, the remaining approaches surveyed by Kühn et al. comprise the *Metamodel for Roles*, the *Integrated Networking Model*, *Data Context Interaction*, and the *Helena Approach* [42]. Additionally, *Macodo* [41] and the *Combined Formal Model for Relational Context-dependent Roles* [56] is included. Since, the run-time aspects of the Helena Approach and Macodo will be discussed in Section 3.2 once again, analysis below is limited to their contributions regarding modeling abstractions.

Metamodel for Roles

Genovese [34] aims to provide a flexible formal model for roles, which is able to capture the basic primitives behind the different roles' notions. He introduces players, roles, and contexts each denoting a subset of a domain of classes or objects on the modeling and instance level, respectively. Classes may have attributes and operations, hence, players, roles, and contexts all have both state and behavior. Players and roles having both state and behavior is a prerequisite for different levels of autonomy of roles. Contexts, *i.e.*, collaborations, with state, behavior, and the possibility to play roles, enable recursive structures of Smart Service Systems. Additionally, Genovese introduces constraints, *i.e.*, any kind of logical rules in order to model different role notions. On the modeling level, players are linked to roles and roles are linked to contexts, hence, the binding is fixed on a level of types. Though the approach provides all required role features, it does not provide a processable specification of an application. Due to the strict linking of players, roles, and context on the modeling level, the open-world semantics is restricted.

	MA 1	MA 2	MA 3	MA 4
Metamodel for Roles	■	■	⊞	□

Information Networking Model

Liu and Hu [61] propose the *Information Networking Model*, which addresses the missing context-dependent nature of roles in the domain of data modeling. Therefore, the concept of *Contexts* was introduced in order to group *Roles*. In the field of data modeling, however, interactions are of no concern. Hence, player, role and context only have attributes but no behavior, which generally limits the approach's applicability to serve as abstraction for engineering class of systems with a high demand of interaction. Consequently, a processable specification is not provided.

	MA 1	MA 2	MA 3	MA 4
Information Networking Model	⊞	⊞	□	□

Data Context Interaction

Reenskaug and Coplien [68] propose the *Data Context Interaction* (DCI) paradigm to point out that *Data* plays a *Role* in *Interactions* encapsulated in *Contexts*. Objects serve as

data containers, implying that objects do not have any behavior. Their behavior is solely defined in roles that, in turn, are part of a certain context. The context manages the binding of role instances to data objects as well as their interaction, which takes place between roles. Evidently, this limits a role's autonomy not to delegate any behavior to its player. DCI is rather a methodological paradigm than a modeling abstraction, hence, it does neither provide a graphical nor textual model to specify a system. Consequently, it does not provide a processable specification at all.

	MA 1	MA 2	MA 3	MA 4
Data Context Interaction	⊞	⊞	□	□

Helena Approach

The *Helena Approach* [42] provides a formal foundation for modeling distributed systems by teaming up roles into ensembles. Ensembles are considered "groups of active entities that collaborate to perform a certain task." [42] The structure of such a collaboration is specified by an *ensemble structure*, which consists of a set of *roles* constrained by multiplicities, and a set of *role connectors* defining the concrete interaction between roles. Hence, ensembles are treated as self-contained units defining a collaboration. Ensemble structures also specify the interaction between roles explicitly, which enforces a specific control flow. Ensembles, however, do neither have behavior nor state in terms of methods and attributes, respectively.

In the Helena Approach, *Components* are considered the player of a role. A component provides basic information usable in all roles the component can play. Though, component types have attributes and operations [42], the latter were never utilized in conjunction with role's operations. The component is only a data container [50], which is why players in the Helena Approach are considered to have no behavior. Roles in the Helena Approach provide the component's functionality essential to an ensemble. A role (more precisely, its type) defines role-specific attributes and operations which are required to store data that are relevant for performing the role, and to fulfill the responsibilities of the role, respectively. Additionally, it defines which component types can contribute the desired functionality to the collaboration and enhances them with role-specific capabilities. Hence, players of a role are restricted based on their type, *i.e.,* the role specifies the component types of entities which are able to play this particular role. Conclusively, the Helena Approach provides HelenaText [51], which is a DSL to specify ensemble structures.

Bucchiarone, Cicchetti, and Sanctis [11] as well as Birdsey, Szabo, and Falkner [8] propose specifications for collective adaptive systems. While Bucchiarone et al. use a notion of roles similar to that of Hennicker et al., the approach by Birdsey et al. relies on agents. Similar to the *Helena Approach*, systems are completely specified, thereby, prohibiting serendipity. Since both approaches are similar to the *Helena Approach* with respect to the requirements posed in the beginning of this section, they are not discussed separately.

	MA 1	MA 2	MA 3	MA 4
Helena Approach	⊞	■	□	■

Macodo

Haesevoets, Weyns, and Holvoet [41] propose *Macodo*, a conceptual model for dynamic collaborations of distributed service-oriented architectures. In Macodo, business processes are functionally decomposed into smaller units in order to improve modularity and reusability of a distributed, service-oriented system.

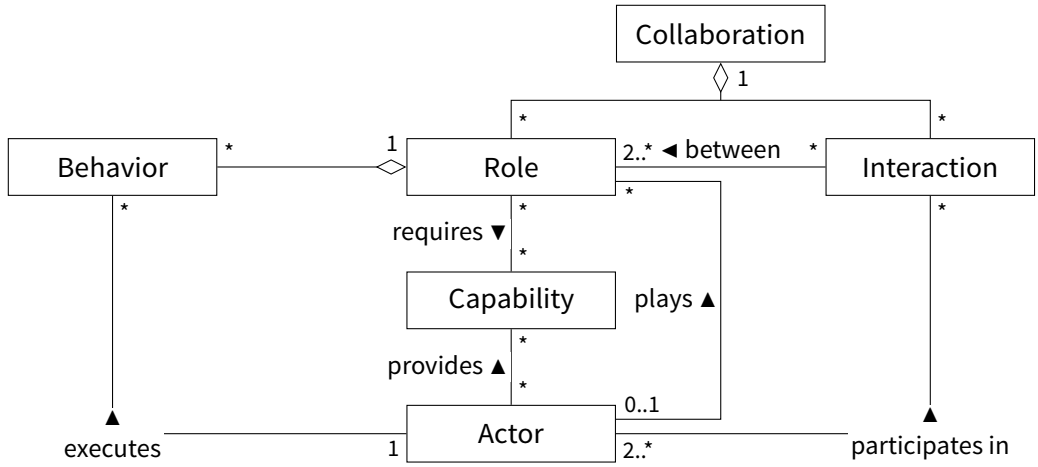


Figure 3.1: Key Concepts of Macodo and their relations. [cf. 41, Figure 2]

Macodo follows the organizational perspective of roles. Its conceptual model, depicted in Figure 3.1, relies on *collaborations*, which are considered a controlled process among a group of actors working towards a common goal. A collaboration has an arbitrary number of both *roles*, representing the different actors and their responsibilities, and *interactions* between the actors of these roles. An actor is the core entity, *i.e.*, the player, which is capable of participating in collaborations by playing roles. In a concrete system, actors are considered business entities, software agents, services, or even people. A

role defines an actor's responsibilities essential to its participation within a specific collaboration. The additional link between actor and interaction in Figure 3.1 indicates that a role not solely depends on the collaboration but also on the actor's participation in a collaboration. This contradicts the organizational perspective, in which a role's existence does not depend on its player's existence. Consequently, this also imposes restrictions on the requirement of serendipity.

Haesevoets, Weyns, and Holvoet [41] additionally define three architectural views each modeling different aspects of the complete system in order to improve modularity and reusability. The *collaboration view* models reusable types of collaborations and their general structure. The structure shows how collaborations are modularized into, *e.g.,* roles, behaviors, and interactions. The *collaboration & actor view* models how collaborations are actually used, *i.e.,* which collaborations are active in the system and which actor is playing which role. The *role & interaction view* models how a collaboration works in detail, *i.e.,* it allows to specify the concrete role and interaction instances in a collaboration, the active behaviors of roles, and how roles delegate the participation in interactions to behaviors. Within these views the overall system is modeled by means of business process modeling, thus, enforcing at least a semi-structured control flow.

Macodo is a design-time approach aiming to improve modularity and reusability rather than addressing variability and flexibility at run time. Collaborations are translated to BPEL processes, actors' and roles' interfaces are described using WSDLs, hence, a processable specification is available but split into several parts and including probably irrelevant information for specifying Smart Service Systems.

	MA 1	MA 2	MA 3	MA 4
Macodo	■	■	⊞	⊞

formalCROM: A Combined Formal Model for Relational Context-dependent Roles

Kühn et al. [56, 57] propose a *Combined Formal Model for Relational Context-dependent Roles*, which aims to integrate the relational and context-dependent (*i.e.,* organizational) in a single, combined model including the ability to constrain role relationships. It is a formally founded descendant of the Compartment-Role-Object Model (CROM) [58].

The core entities in their approach are *Natural Type* (*e.g.,* physical, real-world entity), *Role Type*, *Compartment Type*, and *Relationship Type*. Players are considered a unification of natural types and compartment types, *i.e.,* both physical entities and objectified

collaborations can play roles. Their definition of the *fills* relation implies a strict binding on the level of types, enforcing players to be of several but specific types. A formal definition of the types is not provided, however, at least natural types, role types, and compartment types are assumed to have attributes and methods. However, this strict typing does not allow for serendipity.

The approach strongly follows the 26 classifying features. Though, it does not support all of them, an in-depth analysis reveals that the required role features ⟨1⟩, ⟨16⟩, ⟨19⟩, ⟨20⟩ are supported. Concerning the intertwining behavior of roles and players, their formal model covers some run-time aspects with respect to the *fills* relation but does not provide a statement how behavior can be intertwined. The formal definition reveals that roles are connected using relationships, hence, interaction takes place between roles.

Recently, a corresponding modeling tool was developed [55], which provides graphical modeling support for their formal model and source code generation limited to SCROLL [60], a local run-time for role-based programs, and a python implementation [54] for validating formalCROM models. A processable specification is missing.

	MA 1	MA 2	MA 3	MA 4
formalCROM	■	■	⊞	□

3.1.3 Summary

Table 3.1 on the next page displays the full comparison chart with respect to the requirements MA 1 to MA 4. All selected approaches supported at least the notion of collaborations as self-contained units and roles in each approach are dependent on such. The *Information Networking Model* [61] and the *Data Context Interaction* paradigm [68] primarily address data modeling or capture the context-dependent nature of data in computer programs, respectively. Data, however, do not provide intrinsic behavior, leading to rather static players, which is unsuitable with respect to the posed requirements. The *Metamodel for Roles* [34] and *formalCROM* [56, 57] aim for generalizing or unifying different notions of roles within a common metamodel. This allows to formally specify the overall system using also the perspective on roles of this thesis, presented in Section 2.2, except that players are statically linked on the level of types instead of specifying their shape. Additionally, both approaches do not provide a processable specification.

The *Helena Approach* [42] and *Macodo* [41] are the two most promising approaches. The Helena Approach, however, limits players to be a sole data container, not providing any

		Metamodel for Roles [34]	Information Networking Model [61]	Data Context Interaction [68]	Helena Approach [42]	Macodo [41]	formalCROM [56, 57]
Collaborations	MA 1	■	⊞	⊞	⊞	■	■
1. self-contained units		■	■	■	■	■	■
2. behavior and state		■	□	⊞	□	■	■
3. act as players for roles		■	⊞	□	□	⊞	■
Roles	MA 2	■	⊞	⊞	■	■	■
1. have behavior and state		■	□	■	■	■	■
2. depend on collaborations		■	■	■	■	■	■
3. interaction between roles		■	∅	⊞	■	⊞	■
4. constrainable relationships		■	□	□	□	□	■
Players	MA 3	⊞	□	□	□	⊞	⊞
1. have own state and behavior		■	□	⊞	□	■	■
2. intertwining behavior		∅	∅	□	□	■	∅
3. preservable autonomy		□	∅	∅	∅	□	□
Processable Specification	MA 4	□	□	□	■	⊞	□

■ yes/provided, ⊞ partially/possible, □ not possible, ∅ not applicable

Table 3.1: Detailed Comparison of Role-based Modeling Abstractions for Software Systems.

behavior that can intertwine with a role's behavior. This aspect is covered in Macodo, in which players, referred to as *Actors*, have behavior and are responsible for executing the role's behavior. These actors, however, are predefined, limiting the variability and flexibility at run time. Macodo additionally enforces a partial workflow and utilizes business process models to model the overall system, the latter of which leads to a scattered system specification comprising BPEL processes and WSDL descriptions.

Evidently, none of the presented approaches provides a sufficient abstraction including a respective processable specification to specify Smart Service Systems. All approaches limit the autonomy of roles' players to be of a specific type or to have no individual behavior, contradicting the heterogeneous nature of Smart Service Systems. Extending existing approaches would be conceivable for the Helena Approach or for Macodo. In the Helena Approach, behavior and state are ought to be introduced for both players and collaborations. Additionally, the *role connectors* need to consider the player's behavior in order to realize an intertwined behavior. This, however, would contradict the main goal of the Helena Approach as it would violate the underlying formal model. Concerning Macodo, the strict coupling between *Roles*, *Interactions*, and *Actors* as players of roles within interactions, needs to be dissolved. This would affect the set of *Architectural Views*, which, in turn, would have to be redesigned as well. Finally, this must be reflected in the system specification, which still would be scattered into several parts.

3.2 Role-based Run-Time Systems

The preceding section discussed role-based modeling abstractions to specify Smart Service Systems. The *Concept of Roles*, however, already found its way to run-time systems and even to distributed systems. In this section, role-based run-time systems are analyzed for their utilization to engineer adaptable, collaborative Smart Service Systems. First, a scheme to classify existing approaches is developed, which is subsequently applied to existing role-based approaches including a discussion of their potential utilization within Smart Service Systems. Conclusively, the findings are summarized and open research gaps as well as remaining challenges are derived.

3.2.1 Classification

The requirements posed in Section 3.1.1 have to be extended by run-time aspects of Smart Service Systems derived from the definitions and requirements given in Sections 1.2

and 1.4. In this section, only approaches claiming to rely on a notion of roles are considered. The full scheme is displayed in Figure 3.2 and explained below. The properties highlighted in bold indicate requirements of Smart Service Systems whereas the faded ones, such as non-distributed systems, indicate exclusion criteria.

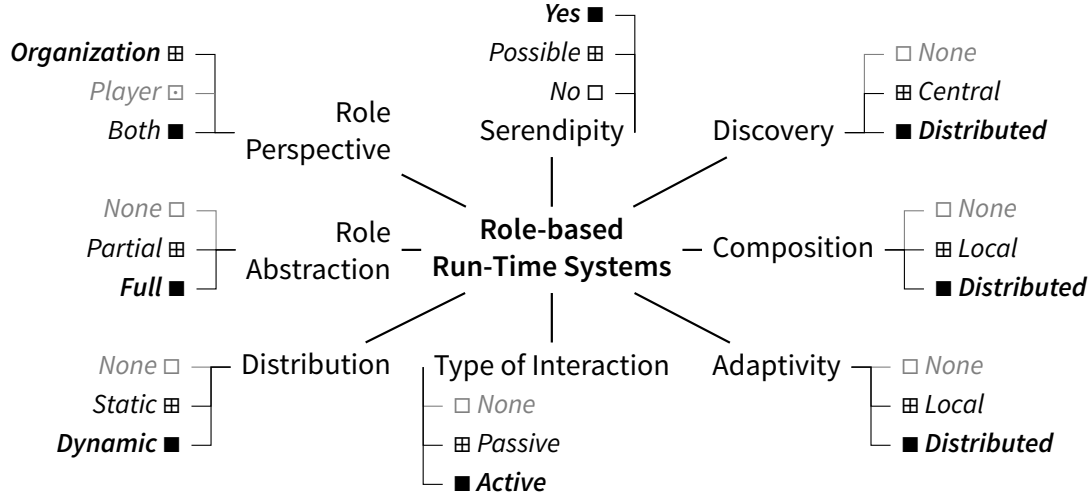


Figure 3.2: Classification Scheme for Role-based Run-Time Systems.

The *Role Perspective* refers to the approaches' perspective on roles [20], *i.e.,* player-centric, organization-centric, or both. The *Role Abstraction* denotes the approaches' support of required role features. In addition to the requirements in Section 3.1.1, following role features that are only applicable at run time are needed: players may play different roles simultaneously ⟨3⟩, players may play the same role several times ⟨4⟩, roles are acquired and abandoned dynamically ⟨5⟩, unrelated players may play the same role ⟨7⟩, and collaborations have their own identity ⟨26⟩. The *Role Perspective* and *Abstraction* jointly capture the support for complex service structures.

Since Smart Service Systems are distributed systems in volatile environments, approaches are distinguished with respect to their degree of *distribution*, which is either *no* distribution, *static* distribution, or *dynamic* distribution, the latter of which is required for Smart Service Systems. Furthermore, in order to achieve automated composition in decentralized environments, individual parts of the system are required to find each other. Hence, the approaches are analyzed whether they provide a *discovery* mechanism, and if so whether it is based on a *central* directory service or discovery knowledge is obtained locally on each of the *distributed* nodes. *Composition*, here, has a twofold meaning: composition is considered *local* if the system is not distributed, but the application consists of modular units at run time; *distributed* if the system is distributed and consists of

modular units at run time, such as SOAs. If the approach does not allow for modularity at run time, composition is *not available*. In this thesis, adaptivity is considered the possibility to reconfigure a system subsequent to its composition. Therefore, it must consist of modular units at run time. If adaptivity is supported, it is to be distinguished into *local* adaptations and *distributed* adaptations. Local adaptations operate on a local knowledge and happen within the boundary of a single system, whereas distributed adaptations affect multiple systems simultaneously. Finally, the *type of interaction* is distinguished as follows: services interact with each other *actively*, or *passively* through a mediator, or not at all. In the former case, a service will invoke a functionality on another service directly, whereas in the latter case a third party will retrieve events or data from a *Service A* and forward it to a *Service B*. Consequently, passively interacting services are not aware of being connected to each other. Finally, the approaches are analyzed concerning their support for serendipity, *i.e.*, to integrate unforeseen entities into the running system. As mentioned, this relates to MA 3-3, *i.e.*, preserving the independence of the player, as the autonomous services are to be developed as independently as possible. A strictly typed relation between the role and its player evidently limits serendipity.

3.2.2 Approaches

Subsequently, various approaches utilizing the concept of roles and dealing with composition, adaptation, or modularity are discussed. Earlier said, Smart Service Systems are volatile, distributed systems, which is why non-distributed approaches [43, 60, 76] are generally excluded from further investigation, even if they support an organization-centric perspective. Nevertheless, non-distributed approaches could be considered a potential local, role-based runtime of an autonomous service.

Since both the *Helena Approach* [42] and *Macodo* [41] were already discussed in Section 3.1.2, only their run-time aspects are addressed herein. Additional approaches included in this investigation comprise *Smart Application Grids* [66], *Role Oriented Adaptive Design* [18, 20, 21], and *Distributed Emerging Ensembles of Components* [13, 12, 49].

Smart Application Grids

Piechnick et al. [65, 66] propose a software architecture, denoted as *Smart Application Grid* (SMAG), in which applications are composed of many small, distributed applications that link to each other dynamically. Dynamically, here, refers to an explicit, extrinsic adaptation based on descriptions similar to event-condition-action rules. The linking

of physically distributed systems, *i.e.,* the process of interconnecting several SMAG runtimes, must be done manually. Hence, distribution is rather static. *Components*, such as *car computer* and *radio*, are stateful, self-contained software modules that denote basic functional elements in SMAGs. Components are described by *ComponentTypes* specifying the component's functional interface by grouping several *PortTypes*. The port type represents an interface description that can be offered or required by a component, and is implemented by one or several *Ports*. Each port type has a unique name and specifies the services of a component providing this interface. Ports are stored in repositories, so that they can be retrieved and integrated at run time.

Role-based modeling is utilized to describe the adaptable structure of SMAG applications at run time as follows: *PortTypes* correspond to role types, hence, *Ports* map onto role instances. The *ComponentType* relates to a class and the *Component* to an object. A collaboration is derived from a pair of provided and required port types, *e.g.,* the car computer component requires a radio controller, which is provided by the radio. Different collaborations consequently provide different ports for the same port types, which leads to variability at run time, such as a different channel list per driver. At run time, a *Component* instance may play roles, *i.e.,* Ports, thereby changing its behavior.

Collaborations capture the three different *Composition Operators* [66, Sec. IV-C], which are *bind*, *filter*, and *adapt*, and denote the collaboration's general structure. These predefined structures restrict the variety of collaborations and incorporate solely two components. Though, Boella and Steimann [9] and Kühn et al. [58] do not explicitly mention how many interacting roles shape a collaboration, predefined collaboration types as the *Composition Operators* are clearly restricting the flexibility of the desired role abstraction. Hence, the SMAG approach does not support complex service structures as a whole. The adaptation is based on event-condition-action rules, which explicitly describe the adaptation (structural recomposition) to be performed. [65]

SMAG's notion of distribution maps onto component-based software, where components are distributed but not necessarily across several computational units. With respect to the proof-of-concept implementation, which relies on Object Teams [43], an organization-centric programming language and runtime, the SMAG approach is considered a rather local approach with manual (static) distribution. Hence, composition and adaptivity are limited to the boundary of a single SMAG runtime.

Smart Application Grids [65, 66]

Role Perspective	⊞	Distribution	⊞	Composition	⊞	Type of Interaction	■
Role Abstraction	⊞	Discovery	□	Adaptivity	⊞	Serendipity	□

Role Oriented Adaptive Design

Colman [18] proposes a *Role-Oriented Adaptive Design* (ROAD) to realize adaptive software systems. ROAD strongly follows the organization-centric perspective of roles and complies more with the characteristics of roles in human organizations, such as a business. Hence, organizations are self-managed composites, consisting of one organizer role and an arbitrary number of functional roles. Organizational roles reconfigure and manage the composite whereas functional roles contain the actual functionality part of the organization's process. The role defines an abstract process, which is to be executed by a player, following the *player-as-executor* pattern.

Roles are interconnected using binary *Contracts*, which are managed by the composite's organizational role and capture relationships as well as functional and non-functional requirements. The structure of these roles shapes the organizational structure. Advan-

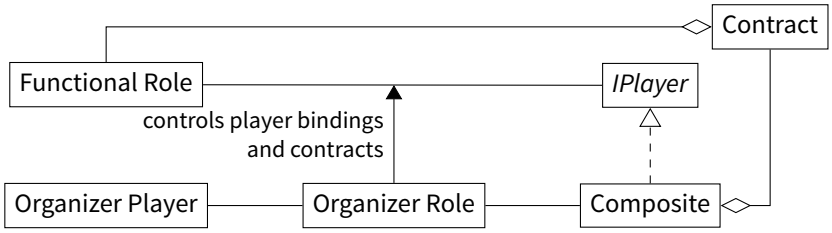


Figure 3.3: Conceptual Relationships in ROAD. [cf. 21, Figure 9]

tages of ROAD's conceptual model, cf. Figure 3.3, are that functional roles and their players are fully decoupled and that every *Composite* adopts a general *IPlayer* interface, hence, the composite itself can act as a player for a functional role in another composite.

The role-based organization structure has two basic adaptation strategies: the first is to restructure the relationships between the roles within the composite, and the he second is to replace a player with an alternative one that better matches the contract specifications. Exchanging players, however, is not achieved automatically but is a task of the organizer role and its player, which are also in charge of evaluating the contracts.

Concerning distribution, web services can be considered players in ROAD. A fully dynamic distribution, however, requires a discovery mechanism, which is of no concern within the general concept of ROAD. [18, p. 214] Instead, discovery is ought to be realized by an organizer role or player, respectively. [18, p. 167] Hence, distribution is considered static as it requires manual intervention. In general, however, it is possible to compose and adapt the system on a distributed level.

Role Oriented Adaptive Design [18, 21]

Role Perspective	⊞	Distribution	⊞	Composition	■	Type of Interaction	■
Role Abstraction	■	Discovery	□	Adaptivity	■	Serendipity	⊞

Helena Approach

The formal foundations of the *Helena Approach* [42] have been discussed in Section 3.1.2. Though it follows the organization-centric perspective, players do not have behavior and a predefined type, which does not allow for serendipity. Nevertheless, the Helena Approach provides an ensemble specification to model distributed, role-based systems. This specification, *i.e.*, HelenaText, can be translated to executable code [51], by means of generated classes representing roles, ensembles, and components.

At run time, the Helena Approach is only conceptually distributed as it maps onto a component-based system architecture [50–52] similar to SMAGs. [66] A fully dynamic distribution additionally requires a discovery mechanism, which is of no concern within the general concept of the Helena Approach. Hence, distribution is considered static. Conceptually, however, it is possible to compose and adapt the system on a distributed level with respect to the restrictions on the desired role abstraction.

Helena Approach [42, 50–52]

Role Perspective	⊞	Distribution	⊞	Composition	■	Type of Interaction	■
Role Abstraction	⊞	Discovery	□	Adaptivity	■	Serendipity	□

Macodo

The design principles of *Macodo* [41] have been discussed in Section 3.1.2. Though it follows the organization-centric perspective, it lacks certain required features, such as the player's autonomy, and provides the desired role abstraction only partially.

The decomposed business processes describe a distributed, service-oriented system. The run-time architecture of Macodo, however, requires central management. [41, p. 23] Macodo focuses on collaborations within a restricted environment, managed by a trusted third party. [41, p. 32] Hence, support for collaborative applications without central control is lacking. Adaptivity in Macodo refers to alternative configurations of the complete system at design time rather than reconfiguring the system at run time.

Macodo [41]

Role Perspective	⊞	Distribution	⊞	Composition	■	Type of Interaction	■
Role Abstraction	⊞	Discovery	□	Adaptivity	□	Serendipity	□

Dependable Ensembles of Emerging Components

Initially, *Dependable Ensembles of Emerging Components* (DEECo) [49, 13] were considered an approach for spontaneously collaborating systems, which are discussed in Section 3.3. Recently they attempted to add a notion of roles [12], which is why DEECo was lifted to this section. Bures et al. [13] categorize DEECo as an Ensemble-based Component System (EBCS), which is defined as a "distributed [system] composed of components that feature autonomic and (self-)adaptive behaviors and are organized into emergent ensembles to achieve cooperation." [13] The term ensemble confers to that defined by Hennicker and Klarl [42] for the Helena Approach.

DEECo, basically, is a component-based system, similar to the Helena Approach or to the SMAGs approach. Components have *knowledge*, *i.e.,* internal state and functionality, and *processes*, which specify the execution of local functionality. Additionally, components adopt *Interfaces*, which describe a partial view on a component's knowledge.

Initially, interfaces solely defined a component's shape by means of its attributes. [49] Later, components explicitly had to adopt the interface [13], introducing a more strict relation. More recently, a notion of roles was added [12]. However, roles are simply a replacement for interfaces without gaining more power.

Ensemble Prescriptions define an ensemble' structure. An ensemble consists of a single coordinator and multiple member components. The ensemble prescription also specifies how data is exchanged between the coordinator and the members. It is noteworthy that a component may be a coordinator or member in several ensembles simultaneously. [49, 13] Moreover, DEECo supports hierarchical structures [12] as ensembles can define a *parent-of* or *child-of* relation to another ensemble (within the same prescription). The ensemble structure, however, is predefined to be a one-to-many relationship.

Technologically, component communication and bindings are extracted from the components' implementation and implicitly specified in the ensemble prescription. A shared middleware exchanges data among the autonomous components, which solely operate on their local knowledge base. Hence, there is neither explicit interaction between the components nor their roles since roles have no behavior. Components are considered to operate autonomously though they are not designed independently from the ensemble.

DEECo, however, supports rather complex service structures as components can join and leave an ensemble at run time. Despite the required interface (or role) a component must adopt, ensemble structures define membership constraints that must be fulfilled by components when they join an ensemble at run time. Concerning distribution and discovery, DEECo components base on OSGi [85] components and discovery relies on OSGi service discovery [13]. The DEECo middleware supports communication among several computational units. [13, Figure 8]

DEECo [13, 12, 49]

Role Perspective	⊞	Distribution	■	Composition	■	Type of Interaction	⊞
Role Abstraction	⊞	Discovery	■	Adaptivity	⊞	Serendipity	□

3.2.3 Summary

This section discussed distributed, organization-centric, role-based runtime systems as enabling technology for engineering, or at least operating, Smart Service Systems. Table 3.2 summarizes the individual classifications of the presented approaches. Evidently, no role-based approach provides a complete solution for engineering Smart Service Systems. Besides distribution, discovery is another prerequisite in order to dynamically compose and recompose such systems at run time. *DEECo* is the first approach addressing all the key requirements of smart service systems, *i.e.*, distribution, discovery, and composition as well as supporting rather complex service structures.

	SMAGs [65, 66]	**ROAD** [18, 20, 21]	**Helena Approach** [42, 50–52]	**Macodo** [41]	**DEECo** [13, 12, 49]
Role Perspective	⊞	⊞	⊞	⊞	⊞
Role Abstraction	⊞	■	⊞	⊞	⊞
Distribution	⊞	⊞	⊞	⊞	■
Discovery	□	□	□	□	■
Composition	⊞	■	■	■	■
Adaptivity	⊞	■	■	□	⊞
Type of Interaction	■	■	■	■	⊞
Serendipity	□	⊞	□	□	□

Table 3.2: Comparison of Role-based Runtime Systems.

However, it is limited to passively interacting subsystems, which might be due to the insufficient utilization of the role concept, and the service structure has a predefined pattern. Components are essential building blocks of their overall system specification and designed to operate autonomously. However, they are not independently designed as autonomous services but strictly related to the ensemble.

3.3 Spontaneously Collaborating Run-Time Systems

So far, role-based modeling abstractions and runtimes were investigated concerning their utilization for engineering Smart Service Systems. Heretofore, none of the presented approaches is capable to dynamically compose and adapt service systems in decentralized environments. To a certain extent, however, smart systems already collaborate as of today, such as a smart phone which is streaming audio or video content to other smart devices without installing specific software or requiring difficult configuration. Whether this also applies to more sophisticated system structures and services, is a question yet to be answered. In this section, the current state of practice for such spontaneously collaborating systems is analyzed. Therefore, the previous scheme, depicted in Figure 3.2 on page 29, is slightly changed as explained below. Subsequently, the approaches are discussed and classified according to that scheme.

3.3.1 Classification

Figure 3.4 depicts the slightly modified classification scheme for *Spontaneously Collaborating Systems*. The *Role Perspective* and *Abstraction* were replaced by the *Service Structure*, explained in Section 1.2 and Figure 1.1. The *Simple* service structure, however, was subdivided into run-time static and dynamic service structures. In contrast to static service structures, dynamic service structures are composed on-the-fly at run time. The remaining criteria are applied as defined in Section 3.2.1. Properties highlighted in bold, again, represent a requirement, faded items are exclusion criteria.

3.3.2 Approaches

Compared to Section 3.1.2, stricter exclusion criteria are applied to approaches that indeed realize spontaneous collaborations of loosely coupled systems at run time. The

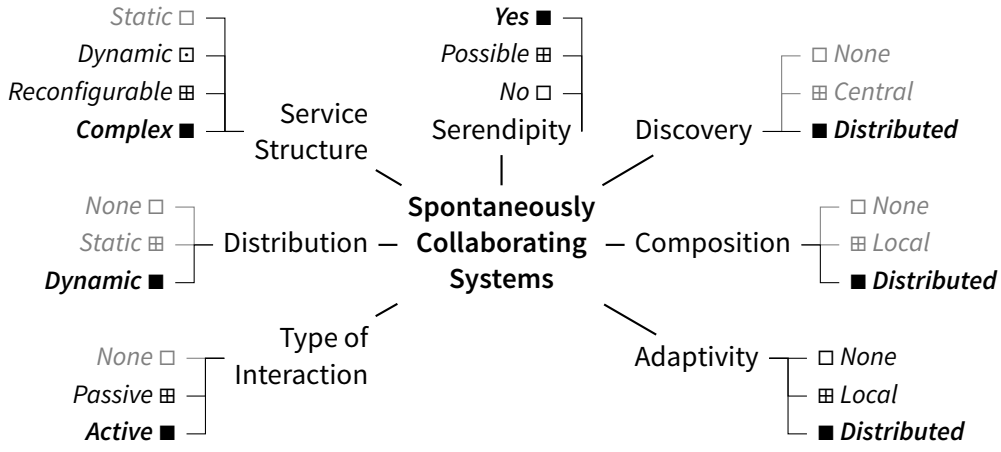


Figure 3.4: Classification Scheme for Spontaneously Collaborating Run-Time Systems.

following approaches have in common that they link at least two rather independently developed systems at run time despite the need for exhaustive manual configuration.

Network Level Protocols

In the smart home, two devices can easily collaborate, such as a smart phone streams audio or video content to a smart tv, or prints a document on a nearby printer. This is achieved by low-level network protocols which all share the same goal of linking two devices dynamically, *i.e.,* a host device, *e.g.,* a smart phone, a tablet, or a computer, is bound to a slave device, *e.g.,* a printer, a smart tv as display, or a radio as loudspeaker.

These protocols can be subdivided into two groups with different application areas. On the one hand, there are protocols to set up multimedia streams, comprising *Universal Plug'n'Play* [63], or proprietary protocols, such as *Apple AirPlay* [45] or *Google Chromecast* [36]. On the other hand, protocols like *Zeroconf* [40] and its descendants *Avahi*[3] and *Bonjour* [10], as well as Microsoft's *Simple Service Discovery Protocol* [35], manage and configure connections with peripheral devices, such as printers.

Both groups share a common basic concept: all devices describe themselves using a standardized device model. These specifications are continuously broadcast across the local network. Additionally, all devices have to implement standardized interfaces, which allow platform- and vendor-independent collaboration. The user triggers the composition process, *e.g.,* by using the scanner and dynamically selecting the target computer or by selecting a target device when playing music on a smart phone. Evidently, the service structure is simple yet dynamic since it is composed at run-time. The limitation of two

collaborating devices is, however, a severe restriction to the applicability of network-level protocols to Smart Service Systems as complex service structures cannot be realized.

Network Level Protocols (NLP) [3, 10, 35, 36, 40, 45, 63]

Service Structure	⊡	Distribution	■	Composition	■	Type of Interaction	■
		Discovery	■	Adaptivity	□	Serendipity	⊞

Self-Organizing Software Systems

Self-Organizing Software Systems (SOSSs), occasionally referred to as Self-Assembling Systems, focus on decentralization and emergent functionality, and usually consist of many interacting subsystems that are either absolutely unaware of or have only partial knowledge about the global system. [70] According to Di Marzo Serugendo et al. [24], the system's structure appears without explicit control or constraints from outside the system. Instead, organization is intrinsic to the self-organizing system and results from internal constraints or mechanisms. SOSSs achieve complex behavior through interactions among individual, autonomous subsystems.

Krupitzer et al. [53] recently investigated the research landscape of Self-Adaptive Software Systems (SASSs), a superset of SOSSs [70], but only a few approaches address decentralized control. This corresponds to Weyns, Malek, and Andersson who conclude that "state-of-the-art self-adaptive frameworks lack support for a growing class of systems in which central control is not an option." [83]

The building blocks of SOSSs are typically services which offer and require functionality through *ports*. The whole system, hence, is composed based on matching provided and required ports, in other words the system's structure is described through simple service dependencies. Fulfilling such dependencies is usually limited to binding one instance of a type to one instance of another type even if multiple instances are available.

MetaSelf Di Marzo Serugendo and Fitzgerald [23] propose *MetaSelf*, a software architecture and development method for engineering SOSSs. The development method consists of four phases tackling requirements analysis, (application) design, implementation, and verification, respectively. The design phase is split into two parts: First, the designer chooses architectural patterns, such as autonomic manager or observer/controller architecture, and adaptation mechanisms, such as governing the interactions and

behavior of autonomous components, *e.g.*, trust or gossip. Additionally, rules for self-organization, and dependability policies are specified. Second, the individual autonomous components, *e.g.*, services or agents, are designed, including necessary metadata and policies. The implementation phase subsequently generates the run-time infrastructure.

Evidently, the development method is a self-contained process leading to a self-contained run-time system that relies on self-describing components, services, or agents and a coordination/adaptation service which is part of the MetaSelf architecture. The system can only adapt within its designed boundaries but is not able to integrate components specified outside the development process. Though parts of the derived run-time architecture can be distributed, MetaSelf will not work in completely decentralized environments. Moreover, it remains unclear how autonomous services are discovered at run time. This would at least require replication of MetaSelf's architectural units, such as the coordination/adaptation service.

MetaSelf [24]

Service Structure	⊞	Distribution	■	Composition	∅	Type of Interaction	■
		Discovery	■	Adaptivity	■	Serendipity	□

FlashMob Sykes, Magee, and Kramer [75] propose *FlashMob*, a formal, decentralized algorithm for distributed self-assembly. Conceptually, *Components* of a certain *Component Type* are instances on certain *Nodes* in a distributed system. Nodes have *State* that represents which component provisions will satisfy which requirements. *State*, hence, describes a system configuration of multiple components.

FlashMob utilizes a gossip protocol [48], which aggregates and distributes state information, to overcome limitations in terms of scalability in order to reach an agreement on a particular system configuration in a logarithmic number of steps with respect to the network size. Each node applies a set of rules after receiving new state information from another node. This includes adding required dependencies, adding provisioned dependencies based on information gathered earlier and evaluating the configuration if it is complete, which means that it meets all the functional requirements specified.

FlashMob [75]

Service Structure	⊞	Distribution	■	Composition	■	Type of Interaction	∅
		Discovery	■	Adaptivity	∅	Serendipity	□

GoPrime Caporuscio et al. [16] propose *GoPrime*, a fully decentralized middleware for utility-aware self-assembly of distributed services. It is an extended version of the Prime middleware [15], utilizing a gossip protocol [47] similar to FlashMob [75]. It is intended to manage distributed systems where a set of peers cooperatively works to accomplish specific tasks. In general, each peer offers services, but could require services offered by other peers to carry out these tasks. The goal is to self-assemble the system among the peers by matching required and provided services utilizing a decentralized approach to network-aware service composition. [17] Services are abstractly described using a joint ontology and implemented using REST [28] interfaces. Moreover, the structure of a system is subject to non-functional requirements, such as performance, dependability, or cost. Hence, *GoPrime* is able to adapt the systems it manages towards the selection, among the set of functionally feasible peers, of an assembly that fulfills global non-functional requirements. Consequently, this leads to a highly reconfigurable and adaptable system, which is still not complex enough as it is not possible to bind multiple service instances of the same type simultaneously.

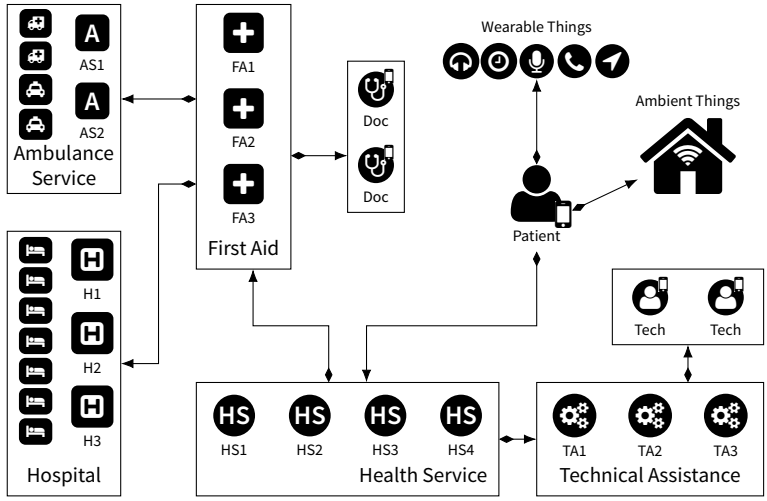


Figure 3.5: GoPrime's Case Study: A Smart Health Scenario. [cf. 16, Figure 8]

Figure 3.5 shows the case study used to demonstrate *GoPrime*'s applicability [16]. The limitation that is clearly visible is that only the *best* service out of a group (a framed box in the figure), such as *FA2* from first aid services, is actively participating in an assembly.

Despite this limitation, *GoPrime* allows for loose coupling, *i.e.*, independently deployed and executed resources; flexibility, *i.e.*, resources can be added and removed into the running application; dynamism, *i.e.*, dynamically discoverable resources are bound into the running application; and – other than all aforementioned approaches – serendipity, *i.e.*, unforeseen resources are integrated into the running application.

GoPrime [15–17]

Service Structure	⊞	Distribution	■	Composition	■	Type of Interaction	■
		Discovery	■	Adaptivity	■	Serendipity	■

3.3.3 Summary

In this section, approaches that realize collaborations of systems on-the-fly at run time were discussed. The approaches were selected on the basis that they interconnect at least two rather independently developed systems at run time despite the need for exhaustive manual configuration. The findings are summarized in Table 3.3, which, for evident reason, also includes *DEECo*, discussed previously in Section 3.2.

Evidently, network level protocols are unsuitable to realize Smart Service Systems. SOSSs provide a good run-time support to achieve spontaneously collaborating systems on-the-fly. *GoPrime* is the most promising approach out of the domain of SOSSs. Generally speaking, a common limitation of SOSSs is the absence of holistic specifications capturing complex service structures. This is due to the emerging nature of SOSSs.

	NLP [3, 10, 35, 36, 40, 45, 63]	**MetaSelf** [23]	**FlashMob** [75]	**GoPrime** [15–17]	**DEECo** [13, 12, 49]
Service Structure	⊡	⊞	⊞	⊞	■
Distribution	■	■	■	■	■
Discovery	■	∅	■	■	■
Composition	■	■	■	■	■
Adaptivity	□	■	∅	■	⊞
Type of Interaction	■	■	∅	■	⊞
Serendipity	⊞	□	□	■	□

Table 3.3: Comparison of Spontaneously Collaborating Run-Time Systems.

DEECo realizes a rather complex service structure at least due to its *coordinator-member* relation, which is intended to be of the type *one-to-many* whereas *GoPrime* is limited to *one-to-one* bindings. Conversely, *GoPrime* achieves serendipity, *i.e.*, it is capable to integrate unforeseen services, which is impossible in *DEECo*.

3.4 Summary

This chapter discussed several approaches concerning their utilization to engineer, *i.e.,* to develop and to operate, Smart Service Systems. First, role-based modeling abstractions were analyzed concerning their applicability to serve as modeling abstraction for Smart Service Systems as well. In general, role-based modeling abstractions fulfill the posed requirements. Consequently, the *Concept of Roles* is a proper abstraction to specify Smart Service Systems. However, all investigated approaches impose restrictions on the *player* entity, such as not to have behavior or state [42, 61, 68] which consequently does not allow the role's and player's behavior to intertwine, or they restrict them based on their type rather than on their shape [34, 41, 56, 57], which prevents serendipity at run time. This is not a limitation of the role concept in general as Boella and Steimann emphasize:

> One of the crucial points […] is the link between role and objects [*i.e.,* the players]. It [*i.e.,* the link,] is annotated by *select from*; this signifies that objects are dynamically selected from a set of relevant objects to play the roles. Many different selection mechanisms can be used. These methods dynamically select the appropriate player objects. In principle, the methods should perform the selection on each call to ensure up-to-date mapping. [9]

Next, distributed, role-based run-time systems were investigated as enabling technology for Smart Service Systems. A classification scheme was developed, which captures the design-time requirements discussed afore as well as additional requirements related to run-time aspects of Smart Service Systems. Most approaches [41, 49, 51, 66] do not comply with the desired role abstraction as they impose type restrictions. Only two approaches, *i.e., ROAD* [18, 20, 21] and *DEECo* [13, 12, 49] provide support for rather complex service structures and their composition. *ROAD*, however lacks dynamic distribution and discovery whereas *DEECo* does not support serendipity and imposes restrictions on service structures mainly due to an insufficient utilization of the role concept.

Finally, non-role-based run-time systems and approaches were investigated that indeed realize spontaneous collaboration of distributed systems at run time. The scheme used afore was slightly change in order to replace the *Role Perspective* and *Abstraction* with the support for *Complex Service Structures*. *DEECo* [13, 12, 49] was included once again as it originally was not a role-based system but was added a notion of roles recently.

Network-level protocols [3, 10, 35, 36, 40, 45, 63] are limited to interactions between two systems. Most approaches in the domain of SOSSs support neither serendipity nor complex service structures as they constitute a mainly linear chain of services [24, 75].

The remaining approaches face the same issues as role-based run-time systems. *DEECo* does not support serendipity, conversely, *GoPrime* [15–17] is limited to reconfigurable service structures. An integration of both approaches is hindered by their contradicting paradigms, *i.e., DEECo* is a component-based system and *GoPrime* utilizes stateless REST interfaces for communication.

Though existing role-based modeling abstractions capture the collaborative nature of Smart Service Systems, they do not preserve their autonomy but impose too many restrictions or lack a processable specification at all. Role-based run-time systems using such specifications close the gap between design and run time but face the same issues as their specifications and consequently lack serendipity and context-awareness. Remaining approaches lack support for complex service structures or discovery. Bottom-up approaches achieving spontaneous collaborations of distributed, autonomous services lack either support for complex service structures or serendipity due to the absence of a specification defining such a complex structure. Thus, it appears that the gap between design and run time and the lack of an autonomy-preserving development methodology are the major challenges for engineering adaptable, collaborative Smart Service Systems.

Conclusively, the key problems of engineering Smart Service Systems, as explained in Section 1.3, can be confirmed. The approach to be developed is consequently required to focus on a specification for complex, context-aware, continuous service collaborations, to decouple the development processes, to solve the discontinuity between design-time specification and run-time composition as well as adaptation, and eventually to achieve on-demand composition of Smart Service Systems.

4 On-Demand Composition and Adaptation of Smart Service Systems

This chapter is concerned with the concepts of the RoleDiSCo approach, which aims to solve the problems stated in Section 1.3. The name RoleDiSCo thereby is inspired from *Role-based Distributed Service Composition*.

Figure 4.1 on the following page provides an overview of the complete RoleDiSCo approach, comprising the development methodology in the upper part and the run-time support in the lower part of the Figure. First, the *RoleDiSCo Development Methodology* separates the development of the overall Smart Service System, *i.e.,* the collaboration, from that of the service and its role essential to the collaboration. The methodology includes a *Role-based Collaboration Specification* to specify Smart Service Systems. Next, the *RoleDiSCo Middleware Architecture* is designed to utilize the artifacts resulting from the development methodology in order to achieve on-demand composition of Smart Service Systems in decentralized environments. The middleware automatically derives discovery information, provides a respective decentralized discovery mechanism and abstracts concrete underlying network infrastructures. Finally, in order to compose and adapt such systems coordinately, a protocol for *coordinated on-demand composition and subsequent adaptation* is proposed.

In the remainder of this chapter, the following scenario is used in order to illustrate the concepts: consider an interactive, tech-enhanced classroom setting in which both the lecturer and its students have smart devices, *e.g.,* smart phones or tablets. The lecturer delivers the lecture using his or her smart device to present accompanying slides. The students' smart devices by contrast are only able to display the current slide and the annotations the lecturer adds to the slides. The students, additionally, are able to give feedback to the lecturer and also may ask questions by virtually raising their hand. A projector in this scenario is considered only a second display attached to the lecture's host, *i.e.,* the lecturer's device. The lecturer is the head of the classroom collaboration and keeps it alive, while students may join and leave during the collaboration.

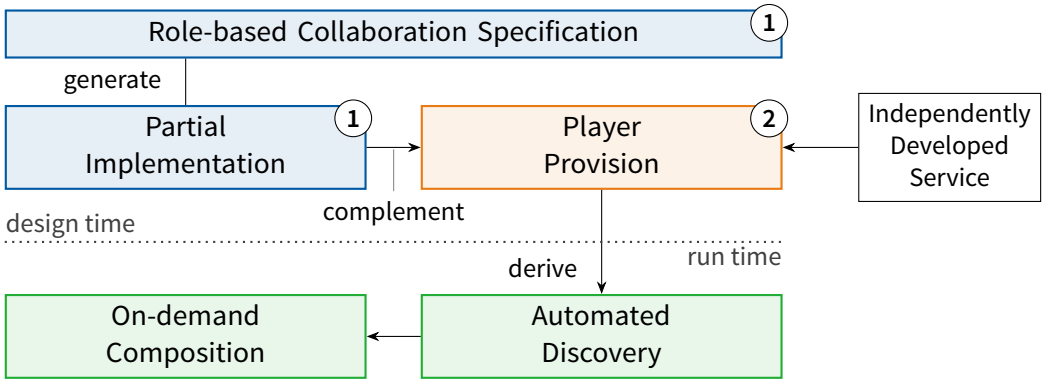


Figure 4.1: RoleDiSCo Development Methodology.

4.1 RoleDiSCo Development Methodology

The *RoleDiSCo Development Methodology* is a two-phase development methodology which allows to demarcate the development processes of the collaboration and its roles, from that of the services. Figure 4.1 depicts the overall process of the approach, which comprises two major design phases of the development methodology.

In Phase ①, a *Collaboration Designer* specifies the application as a self-contained collaboration of roles. Therefore, the designer uses the *Role-based Collaboration Specification*, which will be explained in the subsequent section. Thereof, a *Partial Implementation* is generated. In Phase ②, potentially several developers complement this partial implementation by providing the concrete player implementation or connecting the generated roles to their existing implementation, as explained later. The two phases already contribute to the demarcation of the development processes (RQ 3) since the collaboration designer and the developers of Phase ② do not have to be the same. On the contrary, developers of Phase ② can and shall work independently of the collaboration designer as well as of each other creating several individual player complementations.

4.1.1 Role-based Collaboration Specification for Smart Service Systems

The *Role-based Collaboration Specification* is the essential building block of Phase ①. The requirements posed to the specification are those used in Section 3.1.1 to classify existing approaches. Essentially, the specification shall define a collaboration as a self-contained unit, which in turn defines all the role types enclosed. The collaboration also includes the relationships, *i.e.*, multiplicities and constraints, between role types. The role (type) is required to have state and behavior, *i.e.*, attributes and methods, in order to capture

an abstract functionality. The role's methods should be able to intertwine with the role's player in order to achieve serendipity (RQ 1.3), which is discussed later as well as the roles' ability to interact with other roles. Additionally, the collaboration and its roles must support context-awareness.

A collaboration is assumed to have a *leader* who keeps the collaboration *alive*. In the Classroom collaboration, for instance, the Lecturer initiates the collaboration, which in turn continues to operate as long as the Lecturer and at least one Student are available. However, independent of the number of Students, the collaboration is terminated when the Lecturer is not available anymore.

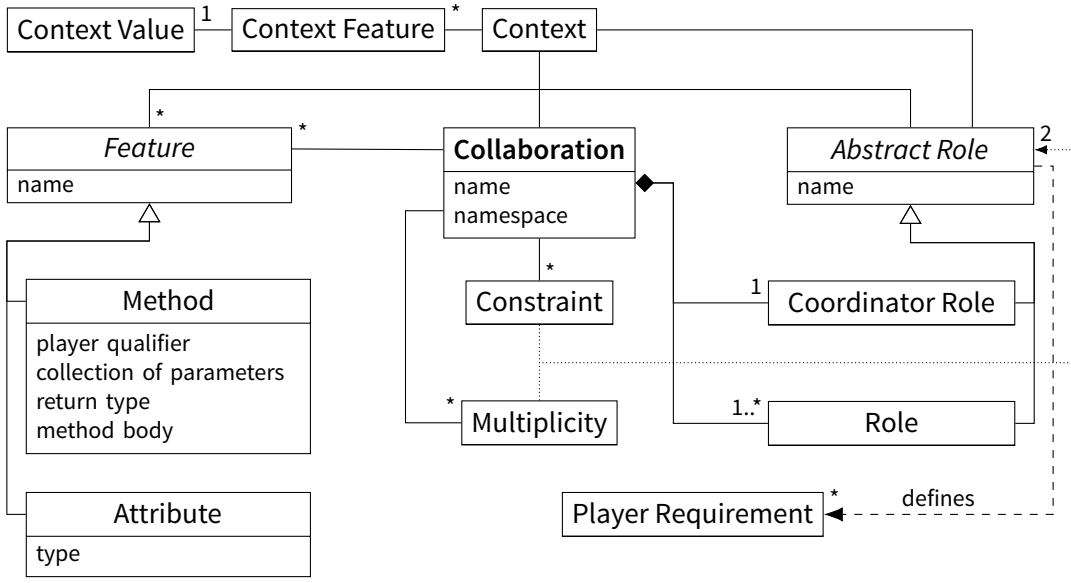


Figure 4.2: Role-based Collaboration Specification's Metamodel.

Figure 4.2 depicts a metamodel based on these requirements as foundation of the specification. In contrast to role-based modeling abstractions presented in Section 3.1.2, the *fills* relation is conceptually relaxed in order to loosen the relation between roles and their players at design time.

Structural Specification

The *Collaboration* founds the core of the specification. Each collaboration must have a unique name within a unique namespace as this will become an essential part of the discovery information at run time. The *Coordinator Role* is distinguished from other *Role*s, both of which are subclasses of *Abstract Role*, as it defines the *leader* of the collaboration,

which, in turn, has exactly one coordinating role. The coordinating role is responsible for initializing the collaboration and keeping it alive. Thus, the role most essential to the collaboration should be chosen as the *Coordinator Role* because its subsystem will be chosen as the *Pervasive Collaboration Coordinator*, which is responsible for managing composition and subsequent adaptation of the collaboration, as it will be discussed in Section 4.3 on page 64. Besides, at least one non-coordinating *Role* must be specified as well. Apart from different names, both types of roles share behavior and structure, however, it is required to distinguish both types in order to enforce that a collaboration specification has exactly one coordinating role type while all others are non-coordinating.

In order to specify the overall structure of the Smart Service System, the roles must be interconnected using *Multiplicities*. Multiplicities are specified as a set of typed pairs containing the role types to be interconnected. The type denotes the multiplicity's kind and must be one of one-to-one or one-to-many. In the Classroom scenario, the link between Lecturer and Student is specified as OneToMany(Lecturer, Student). All relationships are considered to be bidirectional.

Role-playing can be restricted using *Constraints*, such as those introduced by Riehle and Gross [69]. Constraints are specified as typed pairs as well and are applied for two role types (A, B) as follows: *role-dontcare* has no effect on the collaboration at all; *role-implication* ($A \rightarrow B$) requires a subsystem to play role B if it plays role A; *role-equivalence* ($A \leftrightarrow B$) obliges a subsystem to play both roles A and B; and *role-prohibition* ($A \nmid B$) enforces that roles A and B are provided by different subsystems. With respect to the example, Lecturer $\nmid$ Student would cause that a single subsystem never plays both roles in the same pervasive collaboration.

Listing 4.1 on the facing page shows a basic grammar using an ANTLR-like syntax [64], which does not provide a complete language but solely its basic structure. It builds the foundation to derive a concrete programming or domain-specific language (DSL). By intention, *Attribute* and *Method* are not further specified as they depend on grammars of concrete programming languages, which are assumed to provide respective definitions for attributes and methods. Chapter 5 will justify that this is a valid assumption with respect to the research prototype. Constraints and multiplicities are subject to a few restrictions that are not captured by the grammar. The coordinating role exists only once per collaboration, even at the instance level. Thus, the coordinating role cannot be used as the target of a one-to-many relationship. Likewise, a role-implication or role-equivalence resulting in such a relationship is prohibited either. This, however, has to be handled by respective tool support and cannot be easily captured in the grammar.

Listing 4.1: Basic Grammar of the Collaboration Specification.

```
CollaborationSpecification:
    namespace = QualifiedName
    collaboration = Collaboration;

Collaboration:
    type = ValidName
    features += Feature*
    contextFeatures += Context*
    roles += CoordinatorRole
    roles += NonCoordinatorRole+
    constraints += RoleConstraint*
    multiplicities += Multiplicity*;

Feature: Attribute | Method;

Context: 'context' attribute = Attribute;

Role: CoordinatorRole | NonCoordinatorRole;

CoordinatorRole:
    type = ValidName
    contextFeatures += Context*
    features += Feature*;

NonCoordinatorRole:
    type = ValidName
    contextFeatures += Context*
    features += Feature*;

RoleConstraint:
    from = [Role]
    type = (RoleProhibition | RoleImplication | RoleEquivalence)
    to = [Role];

RoleProhibition: '>-<';
RoleImplication: '-->';
RoleEquivalence: '<->';

Multiplicity:
    from = [Role]
    type = (OneToOne | OneToMany)
    to = [Role];

OneToOne: ('one-to-one'|'to');
OneToMany: ('one-to-many' );

QualifiedName: ValidName ('.' ValidName)*;
ValidName: ID;
ID: '^'? ('a'..'z'|'A'..'Z'|'$'|'_') ('a'..'z'|'A'..'Z'|'$'|'_'|'0'..'9')*;
```

Behavioral Specification

Earlier said, roles capture an abstract functionality. Therefore, roles (as well as their surrounding collaboration) may have an arbitrary number of *Features*, *i.e.*, *Methods* and *Attributes*. Attributes consist of a name and a (data) type, such as integer, string, or boolean, and are required in order to capture the internal state of a role or collaboration. Methods consist of a name, a list of parameters and their types, a return type and a method body containing expressions. A model that is able to capture operational semantics of programming languages in order to describe types and expressions is assumed to exist. For instance, Xbase [26] provides a metamodel that captures Java-based languages. The Xtext framework [7, 86], of which Xbase is a part, allow the development of DSLs that result in Java code. Additionally, it is possible to create custom compilers that translate a given model to a particular language.

	with qualifier	without qualifier
method body w/ player reference	partial/full delegation	sole partial delegation
method body w/o player reference	no or full delegation	outbound method call
empty method body	forced full delegation	*not allowed*

Table 4.1: Player's occurrence in a role's method.

In order to incorporate the player's behavior which complements the role's abstract functionality, it is required to specify which parts of the role's functionality are actually delegated to the player. Listing 4.2 shows an exemplary structure of a role's method.

Listing 4.2: Sample Structure of a Role's Method.

```
1 [player] op name(parameters*)[:return type] [{
2   ... // method body
3   player.doSomething()
4 }]
```

The player may appear as qualifier of a whole method in order to fully delegate the behavior or as a receiver of a method invocation inside a role's method body in order to intertwine with the player's behavior. An overview of the resulting behavior is given in Table 4.1. The following cases are to be distinguished:

Forced Full Delegation

If no method body is given, like for an abstract method, and the qualifier is set, incoming method calls (from other role instances) are fully delegated to the player, which

must provide a complementing implementation. This allows to provide different degrees of autonomy, explained by Colman and Han [19].

```
player op methodName(parameters*):return type
```

No or Full Delegation

If a method body without a call to the player is given, and the qualifier is set, incoming method calls are fully delegated to the player if it provides a complementing implementation. Otherwise the role's abstract functionality is used as a default implementation that does not require customizations. Thereby, legacy systems that cannot fully provide a concrete performance can be addressed.

```
player op methodName(parameters*):return type {
  ... // no player call
}
```

Sole Partial Delegation

If the method body contains a player reference and no qualifier is set, only calls to the player inside the method body are delegated to the player, for which a complementing implementation must be provided as well.

```
op methodName(parameters*):return type {
  ...
  player.doSomething(parameters*)
  ...
  value = player.doSomethingElse(parameters*)
  ...
}
```

Partial/Full Delegation

If the qualifier is set and the method body contains references to the player, the whole method is delegated to the player if it provides a complementing method. Otherwise the role's abstract functionality is used and only calls to the player inside the abstract functionality are delegated to the player. Additionally, the role's abstract functionality will be used in case the fully delegated call results in a run-time error.

```
player op methodName(parameters*):return type {
  ...
  player.doSomething(parameters*)
  ...
  value = player.[return type]doSomethingElse(parameters*)
  ...
}
```

Outbound Method Calls

Finally, if no player reference occurs in the method body and the qualifier is not set, the method is treated as an outbound method call. Hence, the role's player can invoke this method in order to communicate with other roles inside the collaboration.

```
op methodName(parameters*):return type {
  AnotherRoleType.doSomething(parameters*)
}
```

The roles' methods can be invoked by any other role within the collaboration if they have a defined relationship. As depicted in Table 4.1, one case is prohibited: an empty method without the qualifier, which would allow for customization through a player.

Dealing with Context & Participation

So far, the specification allows to describe the structure of a Smart Service System based on role types, which allows for composition based on these role types at run time. This, however, would lead to very large collaborations as all systems that provide a certain role type would be integrated. Hence, it is required to restrict participation in a collaboration in a more fine-grained way. As presented in Chapter 2, the concept of roles includes that roles can define requirements for their players apart from method interfaces, which may also include contextual properties. Context is a very generic term that has many definitions, such as the well-known one given by Dey, who describes context as "[…] any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." [22]

This definition, however, is too generic to be captured within the specification. Instead of extending the metamodel to include a comprehensive context model which captures a huge domain of context information, the specification allows to incorporate arbitrary classes as context features. These classes can refer to simple types, such as strings or numbers, as well as to more advanced, external context models. Concerning the metamodel shown in Figure 4.2, a *Context Feature* captures the name and the type of a context information whereas the *Context Value* refers to the concrete value at run time. This allows to use any kind of context model and interpretation within a collaboration. Context features are assumed to be comparable in order to determine whether they are equal or not, in other words, whether two entities are in the same context. With respect to the example in Listing 4.3, the `LectureContext` is a simple data type containing

Listing 4.3: Collaboration Specification with Context Features.

```
1  collaboration Classroom {
2    context LectureContext lectureInformation
3    coordinator role Lecturer {
4      context LecturerContext lecturerPersonalInformation
5      ...
6    }
7    role Student {
8      context LectureContext studentLectureInterests
9      context StudentContext studentInformation
10     ...
11   }
12 }
```

information about a lecture. An exemplary implementation is provided in Listing B.1 on page 149. The collaboration's `lectureInformation` represents the current lecture of an operating collaboration. The student's `studentLectureInterests`, in turn, represents the lecture the student wants to attend. In order to include the student in the collaboration, the two lecture contexts have to be equal.

Additionally, context information is assumed to be serializable, at least in order to determine whether two context features are equal, as described above. Serialized context features will be part of the discovery process, explained later, whereas non-serializable discovery information will be excluded. Please note that, a non-coordinating role type is considered for joining a collaboration only if it is in the same context as the collaboration. In the example, a student's lecture context would have to match that of the collaboration in order to join it. Hereby, only the `LectureContext` feature is considered as it exist for both the Student role and the collaboration. Conceptually, context may depend on the role; its value, however, always depends on the concrete player as it is varying the role's behavior and therefore must be provided by the player.

4.1.2 Derived Partial Implementation

As part of Phase ①, a *Partial Implementation* is derived from the specification and respective source code is generated. Figure 4.3 shows the generated classes for the Classroom scenario. Abstract classes (dotted border) are not generated but part of the RoleDiSCo Middleware, which will be discussed in Section 4.2. These super classes act as common interfaces for all collaboration and role types. For each specified role type, a separate role class, a role interface and its player's optional interface definition

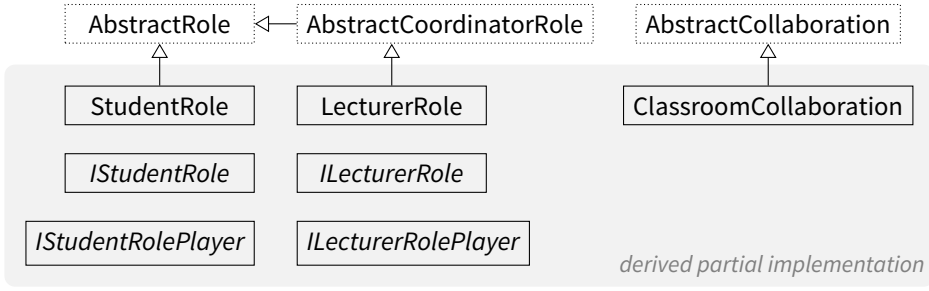


Figure 4.3: Generated Partial Implementation for the Classroom Scenario.

are derived as follows. Attributes inside a collaboration's specification are translated to attributes with respective accessor methods of the role or collaboration classes. Methods are translated likewise. Their abstract functionality is translated to a partial implementation, which must be processed more thoroughly as it affects both the role class and its player's interface depending on the cases shown in Table 4.1. In case of sole partial and partial/full delegation, method calls inside the method body referencing the player are added to the player's interface. The original method call is replaced by calling the middleware's internal dispatch mechanism to delegate it to the actual player at run time. In case of partial/full delegation, an additional check, if the player provides a full replacement of the whole method, is added to the method body's beginning. In this case, the method added to the player's interface is optional and the method call is delegated only if a player instance provides a replacement. In case of forced full delegation, the method body is fully dispatched to the player. An outbound method call is translated to a role-class method which is invokable by the role's player instance through, for instance, a role-based runtime's or the middleware's internal dispatch mechanism. Method calls to other role types are redirected to the dispatcher, which delegates the call to the respective instance(s). Please note that the derived player's interface is provided for convenience but does not need to be stated explicitly as *implemented* by the player as the dispatcher will check the availability of methods and conditionally delegate method calls.

Crucially important is that multiplicities and constraints are included in the derived collaboration class in order to recreate the collaboration specification at run time. This is a prerequisite in order to achieve automated discovery and composition at run time. One sophisticated way is to provide this information as meta data to the class, for instance, by using annotations, as exemplified in Listing 4.4.

Finally, all generated classes are bundled and ought to be complemented by a developer of Phase ②, who does not need to provide corresponding implementations for all roles but only for those of which the corresponding service provides a concrete functionality.

Listing 4.4: Relationships in a Generated Collaboration Class.

```
1 @Constraint(
2   from = LecturerRole, to = StudentRole, constraint = ROLE_PROHIBITION )
3 @Multiplicity(
4   from = LecturerRole, to = StudentRole, multiplicity = ONE_TO_MANY )
5 class ClassroomCollaboration extends AbstractCollaboration { /* ... */ }
```

4.1.3 Player & Context Provision

In Phase ②, potentially several developers complement the abstract functionality of one or several roles and thus provide their concrete performance, *i.e.*, the player(s). The players' optional interfaces technically represent the *player requirements* with respect to method signatures. How a Phase ② developer links the generated roles with existing implementations is part of the case study discussed in Chapter 6. The remainder of this Section explains how the contextual requirements are to be addressed by Phase ② developers and how they provide the appropriate player for a certain context.

Context Provisioning

The necessity to provide context information was motivated previously. At run time, roles initially may only exist as types as they are intended to be instantiated during the composition process for a specific collaboration individually. Hence, they cannot provide context information on their own easily as they only exist as types and therefore could only provide static information that would already be part of the specification. Conceptually, context should be provided by the prospective player, which is, at first, the adaptive subsystem and therein a complementing class, core object, etc., which is provided by the runtime. In order to streamline the context-player issue, context and players are mapped to each other, as depicted in Figure 4.4. Every player might be

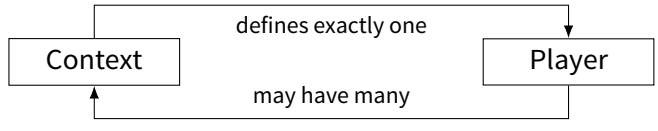


Figure 4.4: Context–Player Relation.

appropriate for several contexts and thus may provide multiple contexts, while each context is mapped to exactly one player. The context provided to instantiate a player and thereby the specified context features are assumed to be precise enough to result in

exactly one player type or instance. In order to delegate context provision to Phase ② developers, they have to implement a *Context Provider*, matching the following interface:

`getContext(role type [, feature]) : context or value`
: This method has to return the concrete context values for the role type's features specified in the collaboration specification's context blocks, *e.g.*, an instance of `StudentContext` for feature `studentContext`. Unless a concrete feature is specified, a set including all feature-value pairs (*i.e.*, `context`) is returned.

Player Provisioning

Though players and context are closely linked to each other, they need to remain loosely coupled. Roles, especially the coordinating roles, do not necessarily have assigned context features, which eliminates the need for a context provider. Thus, player provisioning is a separate task for which the Phase ② developer is asked to implement the *Player Provider* interface in order to connect the derived role classes with an existing player implementation, thereby addressing RQ 1.3, *i.e.*, serendipity, and RQ 2, *i.e.*, the demarcated development processes:

`hasPlayer(role type [, context]): boolean`
: This method determines whether a complementing player implementation is available for a given role type and context. Context, here, refers to a set of all features specified in both the role type's specification and the collaboration's specification.

`getPlayer(role type [, context]): player type or instance`
: This method should provide either a complementing player instance or type that can be instantiated on demand for the given role type and context.

4.2 RoleDiSCo Middleware Architecture

A Decentralized Middleware Architecture for Smart Service Systems

In the previous Section, the intertwined development processes of a service and its role essential to a collaboration have been decoupled using the *RoleDiSCo Development Methodology*. Therefore, a *Role-based Collaboration Specification* was introduced that focuses on the key features of Smart Service Systems, *i.e., complex service structures*, *collaborative nature*, *serendipity*, and *context-awareness*. This Section addresses the run-time support for Smart Service Systems and thereby focuses on the discontinuity between the design-time specification and the run-time composition and adaptation of Smart Service Systems. In order to coordinately compose and subsequently adapt Smart Service Systems in a decentralized environment, a middleware is required to provide an automated, decentralized discovery mechanism (RQ 3.1) as well as mechanisms to derive respective discovery and composition information in an automated way (RQ 3.2).

Figure 4.5 displays the top-level architecture of the RoleDiSCo middleware, which is supposed to exist and run on every node in a decentralized environment. The dashed

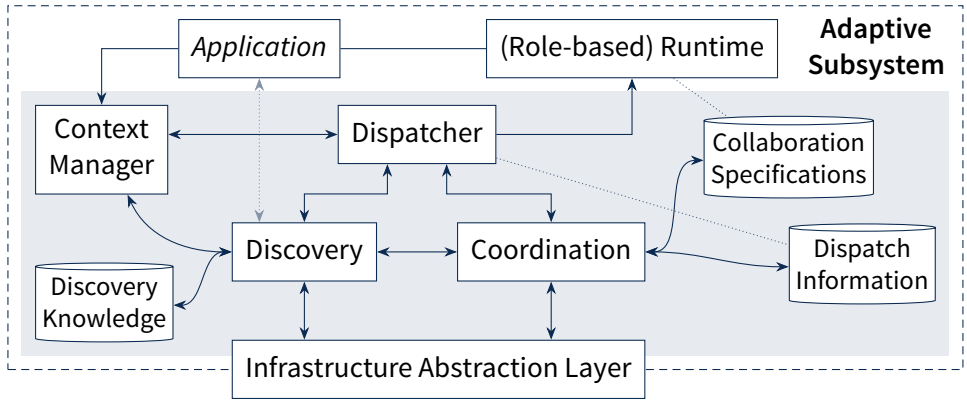


Figure 4.5: High-Level-Architecture of the RoleDiSCo Middleware.

line represents the system boundary of a single subsystem. Nodes communicate via the *Infrastructure Abstraction Layer*, which is introduced in order to completely abstract from concrete underlying protocols and infrastructures (RQ 3.3). The middleware has three major repositories. The *Collaboration Specification* repository contains a representation of locally available collaboration types, role types and their relations. The *Discovery Knowledge* repository is supplied with discovery information, *i.e.,* the collaboration types, the role types and their respective context, as well as the subsystems the role types are located on. These are obtained by the *Discovery* module, which continuously distributes

such information via the infrastructure abstraction layer to other subsystems and collects it from others, respectively. Additionally, the *Context Manager* obtains context features that are specified in the collaboration specifications and required for fine-grained discovery and composition. The *Dispatch Information* repository contains information regarding instantiated collaborations, *i.e.,* the role instances and their respective subsystems. This information is obtained and used by the *Coordination* module, which comprises the *Composition Management* and thereby is responsible for composition and adaptation. The *Dispatcher* module builds the interface to the local, potentially role-based, runtime in order to find and manages collaboration types, role types, and respective players. The remainder of this Section continues with a detailed discussion of each of the modules.

4.2.1 Infrastructure Abstraction Layer

In order to abstract from concrete network protocols and infrastructures, network communication relies on the *Infrastructure Abstraction Layer*. Thereby, every node in the infrastructure is assumed to be reachable via this layer, for instance by using a gossip-based communication approach as proposed by Caporuscio et al. [16] or Sykes, Magee, and Kramer [75], and the approach becomes independent of underlying technologies. The infrastructure abstraction layer defines an interface, described below, for sending and receiving messages to and from the infrastructure. The methods of which the names are printed in italics have to be provided by a concrete implementation of the interface as they depend on concrete network protocols and interfaces.

`publish(message)`
: An implementation of this method must ensure that the `message` is sent to *all* nodes within the infrastructure. It does not expect an immediate reply, thus, it is a non-blocking operation.

`send(target[s], message)`
: An implementation of this method must ensure that the `message` is sent to the target node(s) within the infrastructure. It does not expect an immediate reply, thus, it is a non-blocking operation.

`dispatch(target[s], method, parameters [, return type [, callback]])`
: An implementation of this method must ensure that the `method` call is dispatched to the target system(s) within the infrastructure. If a *return type* is specified, this method expects the remote method to provide a return value, which is then returned to the original caller. A *callback* may be passed, which is executed upon receiving the return value, and results in a non-blocking method call.

`receive(message)`
The messages received through an underlying implementation should be forwarded to the *receive* method, which will continue further processing. The default behavior is to forward the message to its corresponding processor.

`[add|remove]MessageProcessor(message type, message processor)`
This method provides a default behavior to add or remove a message processor for a certain type of messages.

At this point, messages are assumed to be delivered reliably. A concrete implementation is asked to provide reliable messaging as well as request/response handling in order to realize method calls with return values. The *message types* mentioned above will be discussed individually in the remainder of this chapter, each within the section(s) explaining the part(s) of the middleware they are relevant to. In order to send messages to specific subsystems, they require unique addresses, which are generated within the middleware. Therefore, each subsystem generates a unique identifier [59] the first time it is started and keeps that during its lifetime. Thus, a concrete interface implementation must provide some kind of address mapping to translate the unique identifiers used within the middleware to physical addresses required for real network communication.

4.2.2 Context Management

As part of Phase ② of the RoleDiSCo Development Methodology, developers have to implement a *Context Provider*, as explained in Section 4.1.3. In order to manage those providers and to use their information during discovery and composition, the *Context Manager* is introduced, which is responsible for acquiring context information defined in the collaboration specification. It comprises both a repository to cache context information and a module, into which Phase ② developers have to hook into in order to register the context providers. Additionally, the context manager is used to collect context information for broadcasting discovery information. The context manager obtains the context features to be requested from the providers from the derived partial implementation. Providers can also notify the context manager, whose interface is described below, about changes using the `update` method.

`[add|remove]ContextProvider(role type, context provider)`
This method adds or removes a context provider for a certain role type. Context providers have to be registered to the context manager for both pull- and push-based context retrieval.

`update(provider, role type [, feature])`
Registered context providers may push context changes. The context manager will retrieve the context values itself. This may also cause dependent modules, such as the discovery module, to perform some tasks.

`getContext(role type [, feature]): context or value`
This method returns the cached context values for a given feature and a specific player class or delegates the call to the corresponding context provider(s) if no values are cached. If the feature is omitted, a set of key-value pairs is returned. It is used by other modules, *e.g.,* discovery, as context providers are not accessed directly.

4.2.3 Local Repositories & Knowledge

Each subsystem comprises three major repositories as well as a few other data structures to maintain knowledge for continuous operation. The *Collaboration Specifications* repository contains all locally available collaboration specifications, comprising the collaboration class, its role classes, and thereby also the relationships within the collaboration. Collaboration types can be added to and removed from the repository using respective methods. In order to add a collaboration specification to the repository, the collaboration's type, the coordinating role's type and all non-coordinating roles' types have to be passed. Removing a collaboration from the repository only requires the collaboration's type. The repository is complemented with a relation containing potential players for role types. This is equivalent to the *fills* relation at the modeling level, however, here it is populated at run time.

The *Discovery Knowledge* repository contains all role types available in the infrastructure that have a corresponding player to perform that role, the respective subsystem the role type is located on, and its associated context information. This repository, however, only knows *that* the role type has a player on the respective subsystem, but not the implementation or class name of that. Moreover, the repository may contain multiple entries with the same subsystem and the same role type but different contexts. This does not necessarily imply multiple complementing player implementations for a certain role type, instead, a role type might be applicable in several contexts. Context information, by default, only includes context features that are part of both the collaboration and a non-coordinating role type, except the specification contains more fine-grained participation restrictions, causing all (serializable) context features to be contained here. The repository is populated by the *Discovery* module, which is explained in the subsequent section.

The *Dispatch Information* repository captures run-time information of a running pervasive collaboration (cf. Section 1.2 on page 3). Once a pervasive collaboration is instantiated, which will be discussed in Section 4.3, all participating subsystems keep track of that collaboration and its role instances. This allows to dispatch methods between roles. Though only the Pervasive Collaboration Coordinator (PCC), *i.e.*, the initiating subsystem, is in charge of maintaining this information, it is replicated to all subsystems participating in the collaboration. This allows to dispatch methods bypass-

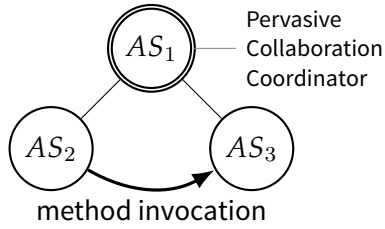


Figure 4.6: Direct Method Invocation, Bypassing the Coordinating Subsystem.

ing the coordinating role's subsystem, depicted in Figure 4.6, in order not to produce unnecessary (network) load. However, if a non-coordinating subsystem notices, for instance, the absence of a required subsystem, it is able to queue the messages until the coordinator orders that subsystem to do something different.

4.2.4 Discovery

The *Discovery* module is responsible for broadcasting and collecting discovery information to and from the infrastructure. Discovery information comprises the role type, as fully qualified name derived from the collaboration specification, the subsystem that type is located on, and the respective context features. Only role types that have a corresponding player are broadcast. Hence, the discovery module is subdivided into local and remote discovery and therefore has to perform different tasks.

Since the derived partial implementation extends classes such as `AbstractCollaboration` (cf. Figure 4.3), the runtime could be introspected in order to discover new collaboration and role types, thereby, populating the *Collaboration Specifications* repository.

The *local player discovery* scans the local subsystem in order to find complementing player implementations for the role types locally available. Therefore, the *Player Provider* interfaces provided by the Phase ② developers are used at first. Assuming that the subsystem's runtime can be introspected and especially in the case of a role-based runtime that it is possible to request complementing player types for role types, additional

mechanisms to find complementing players are provided: First, the local role-based runtime (if available) is introspected and requested to provide the complementing player implementations. In order to deal with heterogeneity and to include legacy systems, a role-based runtime is not assumed to be available in general. Second, configuration files containing pairs of role types and complementing player implementations are processed. Finally, the (generic) runtime is introspected in order to analyze the classes that are annotated as a player complementation. Those classes' annotations also contain the role type the player class can play.

Knowing the local role types that have a corresponding player is a prerequisite for *publishing locally available role types* as only those are to be published that have a player. Subsequently, the discovery information, described above, is broadcast to the infrastructure whenever the infrastructure, the collaboration specifications repository, or the context associated to the role type changes, or a role type has no complementing player implementation anymore, or a complementing player for a previously not playable role type is available. Therefore, for each role type to be published and its respective contexts a `RoleAnnouncement` message containing the role type, the context and the unique identifier of the local subsystem (source address) is broadcast via the infrastructure abstraction layer to all other subsystems. Figure 4.7 shows the message's structure and an example for the Classroom scenario.

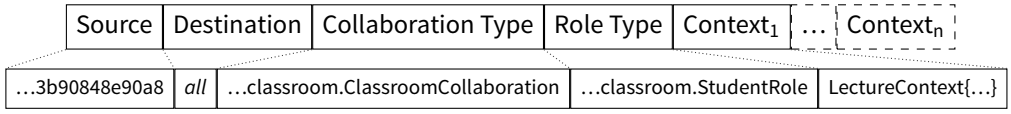


Figure 4.7: Structure and Example of a `RoleAnnouncement` Message.

The *remote role discovery* is the counterpart of publishing discovery information, *i.e.*, whenever a subsystem receives a role announcement message, it the content of that is added to the local discovery knowledge repository if the local subsystem itself contains the discovered role type in the collaboration specifications repository.

4.2.5 Dispatcher

The *Dispatcher* module mainly resolves method calls to remote roles and to local roles' players. Moreover, it is responsible for resolving players for role types, as described for the *local player discovery* before. Therefore, it has to manage the *Player Providers* and has to integrate with the local runtime. Similar to the context manager, the dispatcher is another module, into which Phase ② developers can hook into. However, this needs to

be done only once per runtime, for instance, a specific implementation of the dispatcher is required in order to integrate with a role-based runtime. A fallback implementation is provided in case that no role-based runtime is available.

`dispatchToPlayer(role instance, method name, parameters)`
: This method resolves a call from a remote role and delegates it to the local player. It replaces the player references, originally specified in the collaboration specification, in the derived partial implementation.

`dispatchToRole(calling role instance, target role type, method name, parameters[, callback])`
: This method resolves a call from a local role to a remote role. It is derived from one-to-one relationships specified in the collaboration specification. The calling role instance thereby is used to determine the collaboration and, thus, the target role instances. As method calls may be dispatched in a non-blocking way, an optional callback might be passed that expects the return value of the remote system as input.

`dispatchToRoles(calling role instance, target role type, method name, parameters[, callback])`
: In contrast to the method before, this method resolves a call from a local role instance to several remote role instances, which are determined as before. It is derived from one-to-many relationships specified in the collaboration specification. The callback will be invoked for every received return value.

`hasPlayer(role type[, context])`
: This method is mainly used within the discovery module in order to determine whether a role type has a complementing player implementation. The fallback strategy to answer this question was already explained within the *local player discovery*. Additionally, it is used in the beginning of the composition process in order to double check the availability of a player. In both cases, the *Player Providers*, described in Section 4.1.3, are utilized for this task as well.

`getPlayer(role type[, context])`
: This method asks the local runtime to return an instance of a complementing player implementation for the given role type in an (optional) context. It is used for dispatching as well as binding and unbinding during the composition process. The fallback strategy is to instantiate the complementing player implementation, which was discovered as described afore. The *Player Providers* are utilized for this task similarly to the previous method.

`createRole(role type, context)`
: This method instantiates a role within the given context during the composition phase.

`[un]bind(role instance, player instance)`
: This method binds or unbinds a role in the composition phase. The fallback strategy is to store the binding relation internally. A role-based runtime, however, might choose more sophisticated strategies.

`[de]activate(role instance)`
: This method finally activates a role in the composition phase. After that, method calls are effectively dispatched to the player and to other roles. The fallback strategy is to annotate the respective entry of the binding relation with an *active* flag.

4.3 Coordinated Composition and Subsequent Adaptation

The *Concept Of Roles* was clearly argued to be beneficial to the automated composition and adaptation of Smart Service Systems. However, applying the concept of roles poses some new requirements on the composition and adaptation process: in contrast to service- or component-based approaches, in which the service or the component is available and simply needs to be bound, roles can be instantiated several times in different collaborations. Thus, roles initially appear as types, which have to be instantiated. From the organizational perspective, thus, a role must be instantiated in a given context (referring to that of the collaboration), bound to a player (that is able to play the role in the given context), and eventually that binding must be activated in order to apply the behavioral changes. According to Jäkel et al. [46], the following life cycle states of roles exist at run time: not existent, unbound, unbound player, unbound compartment, bound, bound active, bound passive. Each operation that changes a local role's state potentially introduces failures, which has to be considered in the composition process as well.

Consequently, a protocol to achieve coordinated, role-based composition and subsequent adaptation of *Pervasive Collaborations* in decentralized environments is proposed. The protocol's overall procedure is depicted in Figure 4.8 and described below. The color coding is used to highlight the focus, *i.e.,* run-time monitoring and development of a role-based runtime is not in the scope of this thesis while infrastructure abstraction, negotiation and planning are in partial scope. Hence, the focus is on discovery, as already addressed, and coordinated composition. A detailed explanation of the individual steps follows in the remainder of this section.

A pervasive collaboration's composition process is triggered by an event ①, the origin of which is not further specified. This event is forwarded to the *Composition Management* ②, which is responsible for providing composition plans, *i.e.,* composition structures of

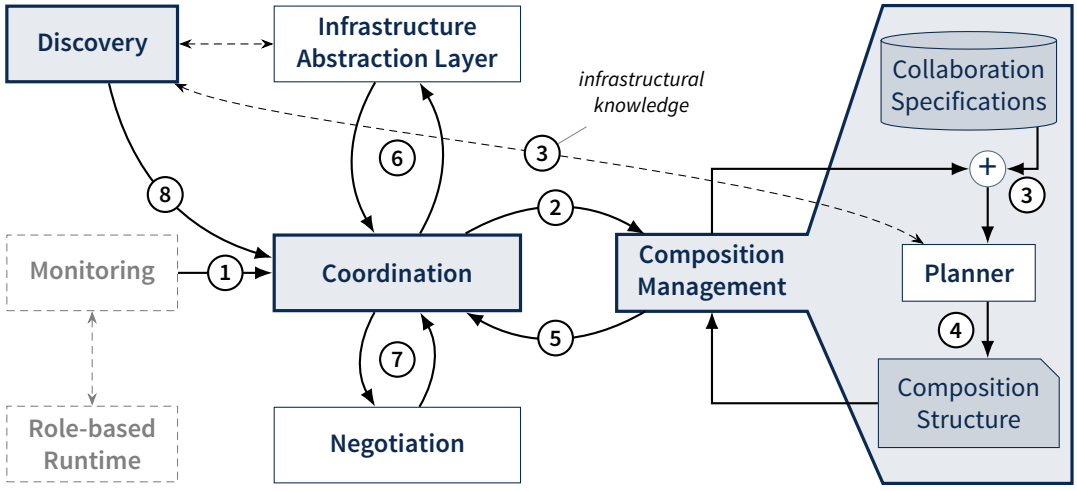


Figure 4.8: Protocol Overview for Coordinated Composition of Pervasive Collaborations.

pervasive collaborations. In Step ③, the appropriate collaboration's specification for the given event is selected from the collaboration specifications' repository and given to the *Planner* component, which enriches the selected specification with infrastructure knowledge obtained from the discovery module. Thereafter, the planner calculates the composition structure ④. The composition management returns the calculated composition plan back ⑤ to the coordination component, which then is responsible for the distributed, coordinated composition process ⑥. This subroutine also includes the handling of run-time failures, such as that a role type on a certain subsystem cannot be instantiated. As long as a revised plan within ⑥ still satisfies the given specification, no recalculation is required. In case of competing pervasive collaborations, a *Negotiation* subroutine ⑦ is expected to find alternative plans or resources (in terms of subsystems). However, disseminating competing collaborations from run-time failures is a research challenge of its own. Hence, an in-depth analysis of negotiation is shifted to future work and the respective subsystem is simply excluded from the collaboration. Thereafter, the composition is set up initially and the subsystem processing the event and setting up the composition gains the position of the PCC within the pervasive collaboration. Step ⑧, which denotes a change in discovery information, may lead to an adaptation of a pervasive collaboration. This causes the Steps ② through ⑥ to be repeated, except that in Step ③ the collaboration specification is already given.

Figure 4.9 exemplifies the operational states of the RoleDiSCo middleware at run time. In the *Bootstrap* phase, all components of the middleware are initialized and locally available collaboration and role types are registered to the collaboration specifications repository. It is succeeded by the *Discovery* phase in which the discovery module

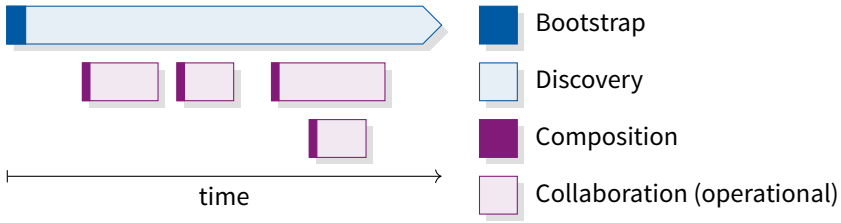


Figure 4.9: Operational States of the Middleware at Run Time.

monitors the local role-based runtime and the infrastructure, *e.g.*, for newly available collaboration and role types, and populates the respective repositories. Each pervasive collaboration is depicted as *Collaboration* phase, which is subdivided into *Composition* and *Operation*. With respect to the Steps described above, Step ① triggers the composition phase, which performs Steps ② through ⑦ before the pervasive collaboration segues into the operational phase, in which Step ⑧ may cause an adaptation.

4.3.1 Initialization and Planning

Figure 4.10 depicts the overall life cycle of a pervasive collaboration, including all intermediate states, on the collaboration-initiating subsystem, *i.e.*, the Pervasive Collaboration Coordinator. The coordination of a single pervasive collaboration is dynamically

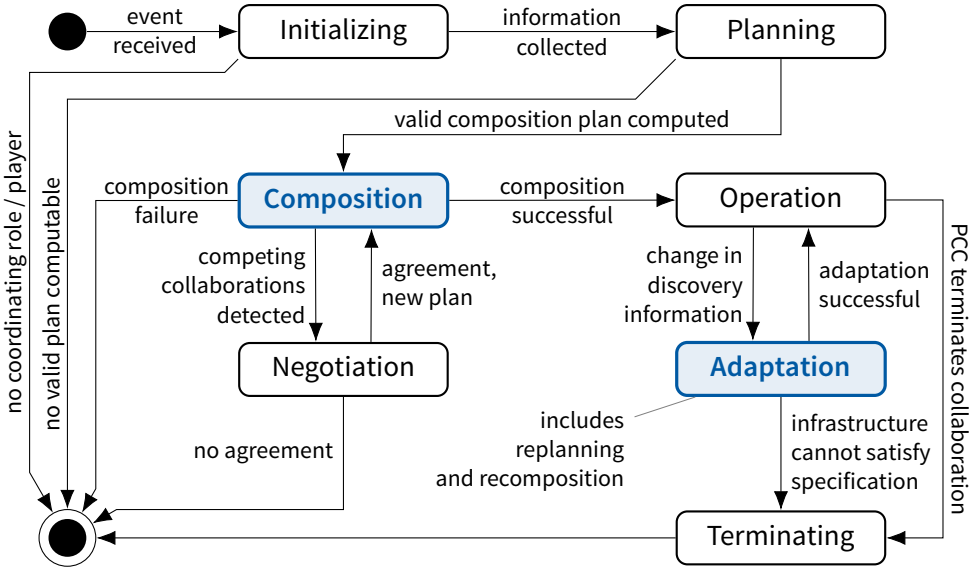


Figure 4.10: Life Cycle of a Pervasive Collaboration.

centralized at run time but not statically predefined at design time. Thereby, decentralized decision-making is avoided while still operating in a decentralized environment.

Once the collaboration has been initiated, it is situated in the *Initializing* state. Every node participating in a collaboration is assumed to provide at least one role, which in case of the PCC is the coordinating role of the collaboration. This role may have been instantiated prior to that point, thereby, initiating the collaboration; otherwise it has to be instantiated within this state and bound to a player. If it is not possible to bind the coordinating role to a player, the composition process terminates immediately.

Within the initializing state, all required information for the planner is aggregated, comprising the collaboration's structure, dynamic context information available because of the aforementioned instantiation, and discovery information, which includes remote context information. Once all information is aggregated, it is handed over to the planner and the life cycle segues into the *Planning* phase. At this point, a general-purpose planner, which allows to match the collaboration specification with provided discovery information in order to create a composition plan, is assumed to exist. The development of such a generic planner was explicitly excluded from the scope of this thesis. If the planner cannot match the provided information with the collaboration's specification and therefore no composition plan can be computed, the process terminates as well. Otherwise, the planner is assumed to provide a composition plan $CP \subseteq (RT \times AS \times Ctx)$, comprising the role types RT, the subsystems AS the types should be instantiated on, and the role type's context information Ctx that was used by the planner and which will be passed to the remote subsystem for further processing. Next, the life cycle turns over to the *Composition* state, in which the distributed, coordinated composition takes place.

4.3.2 Distributed, Coordinated Composition: Coordinating Subsystem

For the sake of simplicity, the *Composition* state was abstracted in Figure 4.10 as it comprises communication with remote systems. Figure 4.11 depicts a complete diagram of the *Composition* state on the PCC's subsystem AS_{PCC}, which is coordinating the distributed composition. Please note that, for the sake of clarity, Figure 4.11 does not contain the *Waiting* states, which are actually entered whenever AS_{PCC} sends messages to other subsystems and waits for their responses. Additionally, the source and target destinations of messages are omitted in the subsequent figures.

The distributed composition is subdivided into four steps, the first of which, *i.e.*, the *Collaboration Initialization*, requires the valid plan of the *Planning* phase and sends a

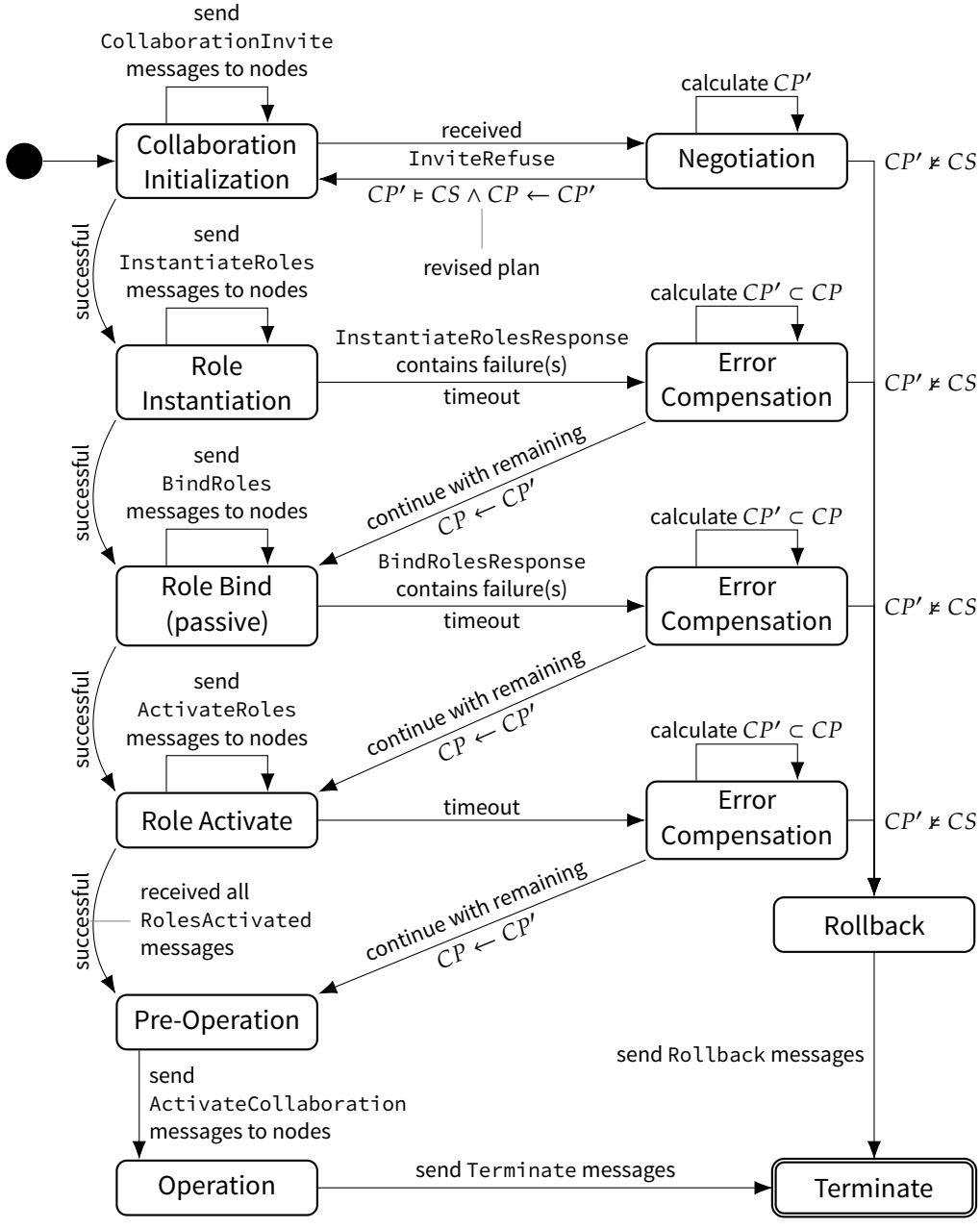


Figure 4.11: Protocol Overview of the Distributed, Coordinated Composition on the PCC's Subsystem.

`CollaborationInvite` message (Figure 4.12) to all subsystems $AS_p \in CP$. The message

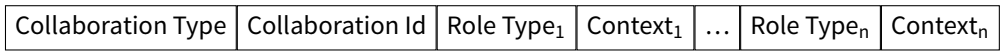


Figure 4.12: Structure of a `CollaborationInvite` Message.

contains the collaboration's type and unique identifier in order to distinguish multiple collaborations of the same type as well as the role types RT and their respective context information Ctx they should be instantiated within. For each participating subsystem AS_p, this information is aggregated within one message and only one message is sent to AS_p. This applies to all subsequent message types as well.

The purpose of this message is to request the availability of AS_p for the initiated pervasive collaboration, *i.e.,* to check whether it is involved in another collaboration and therefore the other collaboration might conflict with the one to be initialized, and to check whether all role types, context information, and respective players can be satisfied or provisioned. Subsequently, the receiving subsystem AS_p knows which role types RT in which contexts Ctx it will have to provide later. Thereby, the message implicitly triggers the composition process on AS_p, which is depicted in Figure 4.19 on page 73. Since competing collaborations and negotiation are discussed later, this aspect is skipped at this point assuming that if AS_p is not able to collaborate, it replies with an `InviteRefuse` message (Figure 4.20 on page 76), or with an `InviteAcknowledgement` message (Figure 4.13) otherwise.

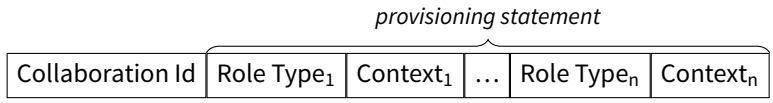


Figure 4.13: Structure of an `InviteAcknowledgement` Message.

The `InviteAcknowledgement` message must contain a non-empty subset of the requested role types and contexts as a provisioning statement. AS_{PCC} waits until it receives responses from all subsystems AS_p. If the provisioning statement only contains a subset of the requested role types, the planner is first asked to check whether this reduced provision still satisfies the collaboration's specification. If it does, the composition process continues with the reduced composition plan CP'. Otherwise, or if the responses contain at least one `InviteRefuse` message (Figure 4.20 on page 76), the *Negotiation* state will be entered, which is subject to further discussion in Section 4.3.4, but eventually results in either a revised composition plan or a rollback of the composition process.

In the *Role Instantiation* state, the nodes AS_p are triggered to instantiate the roles of types RT within the respective contexts Ctx. Therefore, an `InstantiateRoles` message

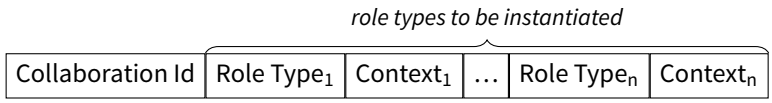


Figure 4.14: Structure of an `InstantiateRoles` Message.

(Figure 4.14), containing the role types and their respective contexts they should be instantiated with, is sent to each subsystem AS_p.

The subsystem AS_p responds with an `InstantiateRolesResponse` message, whose structure is depicted in Figure 4.15. It comprises both successfully instantiated roles and unsuccessfully instantiated role types, as explained in the subsequent section. Given

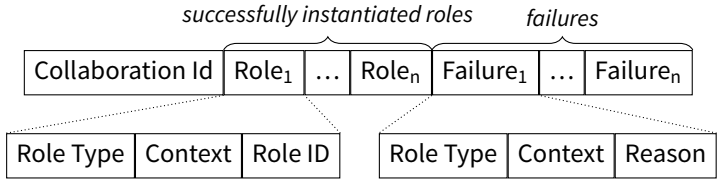


Figure 4.15: Structure of an `InstantiateRolesResponse` Message.

that no failure is reported back, the composition process immediately segues into the *Role Binding* state. Otherwise, if the `InstantiateRolesResponse` message contains at least one failure case, the *Error Compensation* state is entered. Therein, the received failures $E \subseteq CP$ are processed and the role types that could not be instantiated are temporarily removed from the composition plan CP, *i.e.*, $CP' = CP \backslash E$. After that, the planner is asked to analyze whether CP' still satisfies the collaboration specification CS, which is denoted as $CP' \vDash CS$. If CP' does not satisfy the specification, *i.e.*, $CP' \nvDash CS$, the composition process segues into the *Rollback* state causing *all* subsystems AS_p to turn into the rollback state as well in order to rigorously terminate the composition process. Therefore, a `Rollback` message (Figure 4.16), containing the pervasive collaboration's unique identifier, is sent to all involved subsystems. Otherwise, if CP' still satisfies the

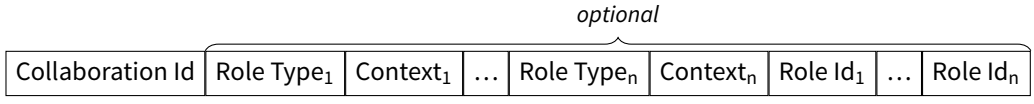


Figure 4.16: Structure of a `Rollback` Message.

collaboration specification, *i.e.*, $CP' \vDash CS$, the temporary changes become permanent ($CP \leftarrow CP'$) and only the failed subsystems are issued a `Rollback` message, which includes the role types and their respective contexts in order to terminate the remote composition process (at least for the particular role type).

In the *Role Binding* state, the subsystems AS_p are requested to bind the roles to their respective complementing player implementation. Thereby, the composition process is able to detect and resolve run-time failures before the pervasive collaboration is turning over to the *Operation* state. Therefore, they are issued a `BindRoles` message (Figure 4.17). The subsequent failure handling and error compensation is equal to

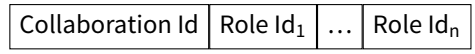


Figure 4.17: Structure of a `BindRoles` Message.

that of the *Role Instantiation* state. The corresponding `BindRolesResponse` message is depicted in Figure 4.18. Finally, the process segues into either the *Role Activation* or the *Rollback* state, as described before.

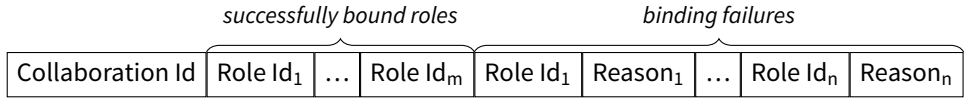


Figure 4.18: Structure of a `BindRolesResponse` Message.

In the *Role Activation* state, the collaboration is assumed to be composed validly as all run-time failures should have occurred before. This state is some kind of synchronization point among all participating roles within a pervasive collaboration. With respect to the life cycle states of roles at run time, the roles currently are in the *bound passive* state, which means that they do not yet interfere with the player's behavior. Subsequently, all subsystems AS_p are issued an `ActivateRoles` message which has to be replied with a `RolesActivated` message in order to activate the roles and thereby the collaboration.

A missing `RolesActivated` response indicates that the respective subsystem left the infrastructure. Please note that leaving the infrastructure and therefore the collaboration or its composition process immediately terminates that subsystem's process. This also applies to previous states, so that after a certain timeout the respective *Error Compensation* state is entered. The failure handling is similar to that explained before, except that missing subsystems are not issued any messages. As reliable messaging is assumed, the system that did not reply is evidently gone. Consequently, the composition plan is recalculated as described and checked whether it satisfies the collaboration's specification. If it does not, the composition process enters the *Rollback* state resulting in the consequences mentioned afore.

Unless no `RolesActivated` response is missing, the AS_{PCC} segues into the *Pre-Operation* state, in which a last `ActivateCollaboration` message is issued to all nodes AS_p,

including dedicated dispatch information for each AS_p, which is required in order to enable any AS_p to invoke a method on another AS_p directly, bypassing the AS_{PCC} as depicted in Figure 4.6 on page 61. The message itself also serves as an acknowledgement to segue all AS_p into their operational state. A lost subsystem in the *Pre-Operation* state will be handled by a subsequent adaptation, discussed in Section 4.3.5. Eventually, the composition process segues into the *Operation* state.

4.3.3 Distributed, Coordinated Composition: Non-Coordinating Subsystem

In this Section, the composition process is considered from the perspective of the non-coordinating subsystem, in the preceding section often referred to as AS_p. Figure 4.10 depicts the overall life cycle of a pervasive collaboration, including all intermediate states, on the event-processing subsystem, *i.e.,* the PCC. In Figure 4.19 on the facing page, instead, the process is visualized from the perspective of a non-coordinating subsystem. Please note that in case of network failures, such as a disconnect or network isolation, the collaboration always reverts local changes or terminates its participation within a pervasive collaboration. For the sake of clarity, these transitions have been omitted. Additionally, in all waiting states (including the pre-operational state), a time-out causes the process to be rolled back. A timeout may occur if the network connection is lost or the AS_{PCC} node is gone. The *Wait* states, moreover, respond to `Rollback` messages (Figure 4.16 on page 70) segueing the systems into the *Rollback* state.

In the *Remote Initialization* state, the pervasive collaboration is instantiated locally. If competing collaborations are detected or other reasons, such as limited resources or resource locks, prevent the local system from joining the collaboration, an `InviteRefuse` message (Figure 4.20 on page 76) is sent back and the *Rollback* state is entered. Otherwise, an `InviteAcknowlegdement` message, including a concrete provision of role types and respective contexts, is sent back to the AS_{PCC} and waits until an `InstantiateRoles` message is received before the system enters the *Role Instantiation* state.

In the *Role Instantiation* state, the previously explained *Dispatcher* triggers the role-based runtime or the respective fallback mechanism, to instantiate the roles of the types given in the `InstantiateRoles` message. Therefore, it uses the context Ctx assigned to each role type RT during the *Planning* phase in order to find or create the right player implementation on AS_p. If no player is existent yet, it should be instantiated at this point as well. Though only role types that have a complementing player should have been considered for instantiation, roles without a complementing player implementation locally available at this point should not even be instantiated.

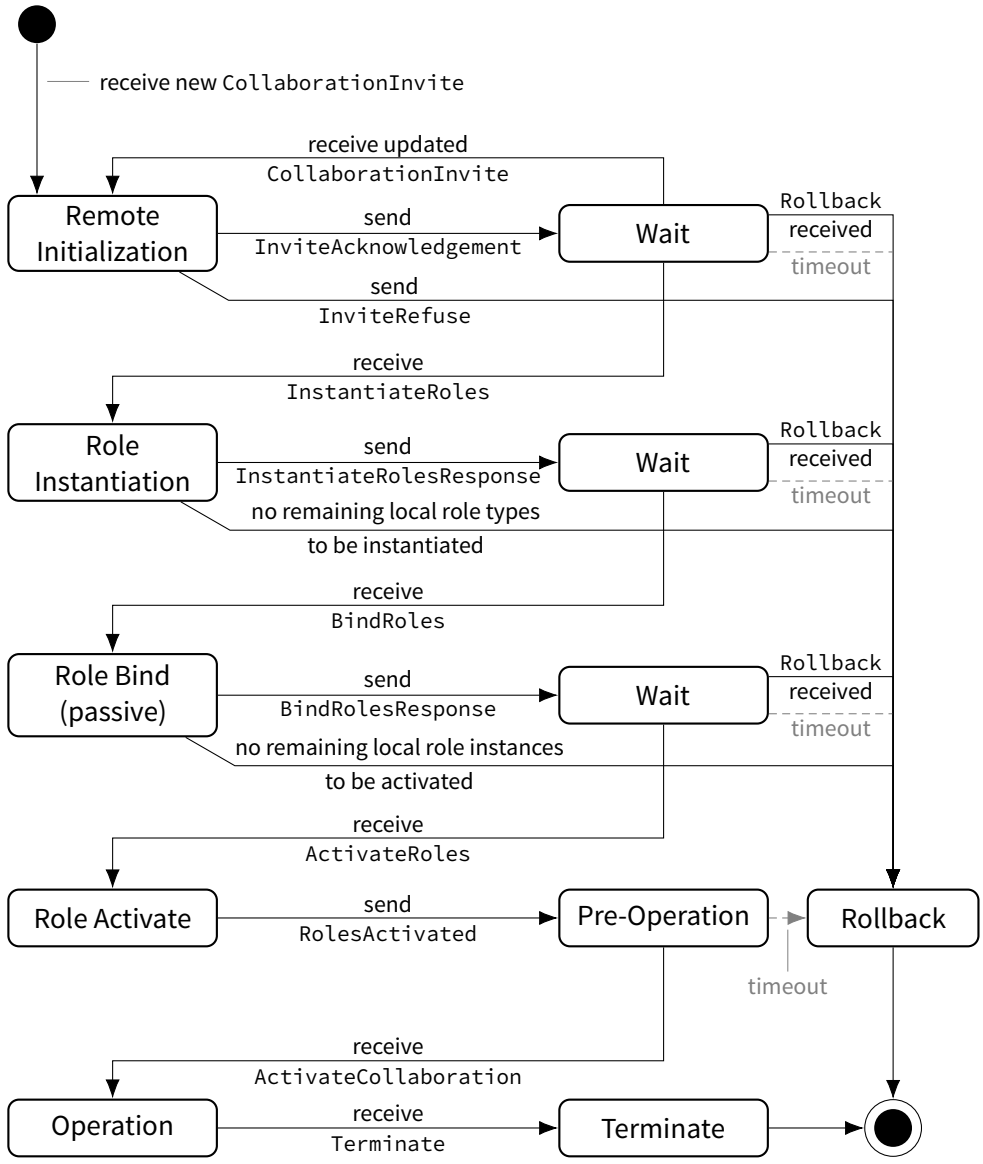


Figure 4.19: Life Cycle of a Pervasive Collaboration on a Non-Coordinating Subsystem.

The results of this step are accumulated and reported back to AS_{PCC} within one `InstantiateRolesResponse` message. Provided that an instantiation was successful, the instantiated role's type, its respective context, and its unique identifier representing the role instances are added to the message's set comprising successfully instantiated roles. If the instantiation was not successful, AS_p is assumed to have made every effort in order to instantiate the role but finally failed and any other attempt would fail as well. In that case, the role types which failed to be instantiated, their respective contexts, and optional reasons are added to the set failures, also part of the `InstantiateRolesResponse` message. Consequently, a waiting state is entered until a `BindRoles` message is received, or, if no other role types remain, the local system rolls back and terminates the process.

In the *Role Bind* state, the dispatcher is asked to bind the roles to their players. The dispatcher delegates this task to the role-based runtime or uses the respective fallback mechanism, *i.e.,* the dispatcher instantiates the player and keeps a record of the binding. Similar to the `InstantiateRolesResponse` message, the results are accumulated and reported back within a single `BindRolesResponse` message. Therefore, the unique identifiers of successfully and unsuccessfully bound roles are added to the message's set comprising the respective responses. The failure handling is equal to that before.

Finally, the *Role Activate* state is entered. The dispatcher is ordered to *activate* the roles, so that they effectively interfere with their player's behavior. This operation is assumed to never fail locally, as it simply enables the dispatch between the role and its player and vice versa. Furthermore, any failures related to the instantiation or binding of a role and its player are assumed to having occurred before. Consequently, the activation is reported back in a `RolesActivated` message, and the system segues into the *Pre-Operation* state. As soon as all participating subsystems have activated their binding, the AS_{PCC} issues the `ActivateCollaboration` message, which also includes concrete dispatch information, the local AS_p needs in order to interact with other roles. Only a timeout may cause the *Pre-Operation* state to roll back. Otherwise, it turns over to the *Operation* state until the collaboration should terminate properly.

4.3.4 Competing Collaborations & Negotiation

In the preceding sections, the challenge of competing collaborations and how to resolve them has been skipped. Two independent collaborations requiring the same resources are considered *competing collaborations*. Resources in that context may be roles, their players and especially resources on the subsystems acquired by the players. In Chapter 6, a scenario of a Distributed Slideshow is introduced, in which a nearby device is used

spontaneously to display some pictures. Now imagine, a legacy projector was included, which would only operate in one pervasive collaboration simultaneously. Hereinafter, the basic negotiation process, part of the composition protocol, is outlined.

Negotiation is only necessary when at least one pervasive collaboration is in the process of composition, *i.e.,* it has not reached the *Operation* state yet. Consider two subsystems, AS_{PCC}, the initiating subsystem, and AS_p, one of the participating subsystems. Initially, the following cases have to be distinguished for AS_p since AS_{PCC} has definitely not reached the *Operation* state yet:

i) AS_p is not involved in any pervasive collaboration
ii) AS_p is involved in a pervasive collaboration as a coordinating subsystem
iii) AS_p is involved in a pervasive collaboration as a non-coordinating subsystem
iv) AS_p is involved in the composition process as a coordinating subsystem
v) AS_p is involved in the composition process as a non-coordinating subsystem

The first case obviously does not cause competing collaborations, as no other collaboration is running or initiated. In the second and third case, the collaboration is already operating, thus, if AS_p cannot provide the requested resources, as in the example stated earlier, an adaptation of the operating pervasive collaboration would be required. This, in turn, requires a negotiation process between AS_{PCC} and AS_p in the second case or between AS_{PCC} and the PCC of AS_p in the third case. According to Weyns et al. [84], decision-making, *i.e.,* negotiation, in decentralized environments is a challenging task of its own. Moreover, sophisticated negotiation will also involve some reasoning tasks, which makes it another research topic of its own. Consequently, these challenges cannot be addressed entirely within the scope of this thesis. Thus, in the current approach, negotiation in the second and third case simply returns an `InviteRefuse` message and the composition process initialized remotely on AS_p terminates. As a result, the fourth and the fifth case are handled the same way, which means that AS_p itself decides whether it is able to provide the requested resources or not, and subsequently replies with either an `InviteAcknowledgement` or an `InviteRefuse` message.

In a more sophisticated negotiation process, the fourth and fifth case may be decided based on their respective composition's progress. In the fourth case, AS_{PCC} is definitely in the *Pervasive Collaboration Initialization* phase whereas AS_p may be in a state prior or subsequent to that. The pervasive collaboration which progressed more precedes the other one. In the fifth case, it is obviously the composition process on AS_p that is more progressed as it already must have received an invitation.

So far, two subsystems with each a single composition process were considered. A subsystem, however, may participate in several pervasive collaborations. Therefore, the results of the steps mentioned afore have to be aggregated for all pervasive collaborations and composition processes on AS_p. The aggregated results have to be evaluated in order to reply either with an `InviteAcknowledgedment` or an `InviteRefuse` message. Moreover, several AS_p may be part of the composition process of AS_{PCC}. Thus, all `InviteRefuse` messages are collected. In this approach, the participants contained in these messages are removed from the composition plan. In general, a negotiation process could be started among all the participants' coordinating subsystems in order to find the best suitable composition for all competing collaborations.

Figure 4.20: Structure of an `InviteRefuse` Message.

4.3.5 Subsequent Adaptation

In the current approach, adaptation is considered a structural reconfiguration of a pervasive collaboration, which may occur whenever a new node joins the infrastructure, a collaborating node leaves the infrastructure, or a role's context changes. Whenever such an event is detected, the collaboration's state changes to *Adaptation*, in which the adaptation is calculated and performed and no other adaptations are processed, *i.e.*, changes in infrastructural knowledge during that time are kept and processed later. Moreover, adaptation is a transition from a (valid) composition plan CP_0 to a new, valid composition plan CP_1. It is, however, not a structural change in the specification. Adaptations that may result from the events mentioned are the addition ($CP_1 \backslash CP_0$) and removal ($CP_0 \backslash CP_1$) of role instances, with respect to Figure 4.21.

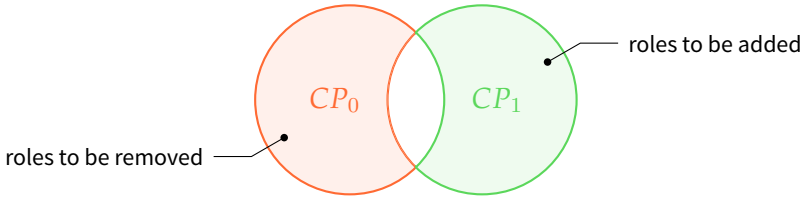


Figure 4.21: Relation of the Composition Plan CP_0 and its Adapted Version CP_1.

The planner is assumed to calculate the composition plan CP_1, Subsequently, the differences between CP_0 and CP_1 can be derived. Thus, deriving an explicit migration of roles between subsystems is conceivable. Weißbach and Springer [82] propose an

approach for fully decentralized execution of adaptations and use a notion of roles as system abstraction. Their approach includes a migration of state and behavior, *i.e.,* the role, which would complement the ideas presented hereinafter. Though an integration of both approaches was explored [MW6], the specification in the current approach does not capture explicit instructions on when to migrate roles and when to only remove a role and add it somewhere else.

An adaptation does not interrupt the operational state of the pervasive collaboration. Instead, it solely prevents the pervasive collaboration from processing several adaptations simultaneously. Processing several adaptations simultaneously introduces conflicts, as two adaptations may interfere with each other, thus, the adaptation may result in two composition plans $CP_{1a} \neq CP_{1b}$ that have to be merged, which is not possible if the two plans conflict with each other. In order to avoid this situation in an anyhow challenging environment, adaptations are performed sequentially in isolation. This, however, does not imply that an adaptation always encompasses only the addition or removal of a single role. Assuming that the network was partitioned, discovery information could be batched, resulting in several role additions and removals comprised in one adaptation.

The *Adaptation* state depicted in Figure 4.10 on page 66 was abstracted for the sake of clarity, similar to that of the *Composition*. Both the adaptation and the composition state are quite similar but have subtle differences, as depicted in Figure 4.22 on the following page. While the *Composition* state started with a fixed composition plan, the *Adaptation* state has to compute a new plan CP_1 in the *Planning* phase. If the new plan is equal to the prior one, *i.e.,* $CP_1 = CP_0$, no further action is required and the adaptation phase terminates. In case that no composition plan $CP_1 \vDash CS$ could be computed, meaning that the infrastructure cannot satisfy the collaboration specification anymore, the pervasive collaboration segues into the *Terminate* state. At this point, CP_0 evidently does not satisfy the specification neither as it would have to be a subset of CP_1 otherwise. For the case that a valid composition plan CP_1 could be computed, `CollaborationInvite` messages are sent to the subsystems AS_p to be integrated, similar to the composition process. If a subsystem AS_p refuses to collaborate or does not reply, CP_1 will be recomputed. The subsequent process is similar to the composition process explained in Sections 4.3.2 and 4.3.3 for each the coordinating and the non-coordinating subsystem respectively.

There are, however, a few differences. The messages sent in the *Invitation*, *Role Instantiation*, *Role Binding*, and *Role Activate* states are only sent to systems that will add a role to the collaboration. $CP_1 \backslash CP_0$ denotes the roles to be created as they are not part of the original composition plan CP_0. The *Error Compensation* states calculate a composition plan CP'_1, which excludes roles failed to be created, from $CP1$. Though it may happen that

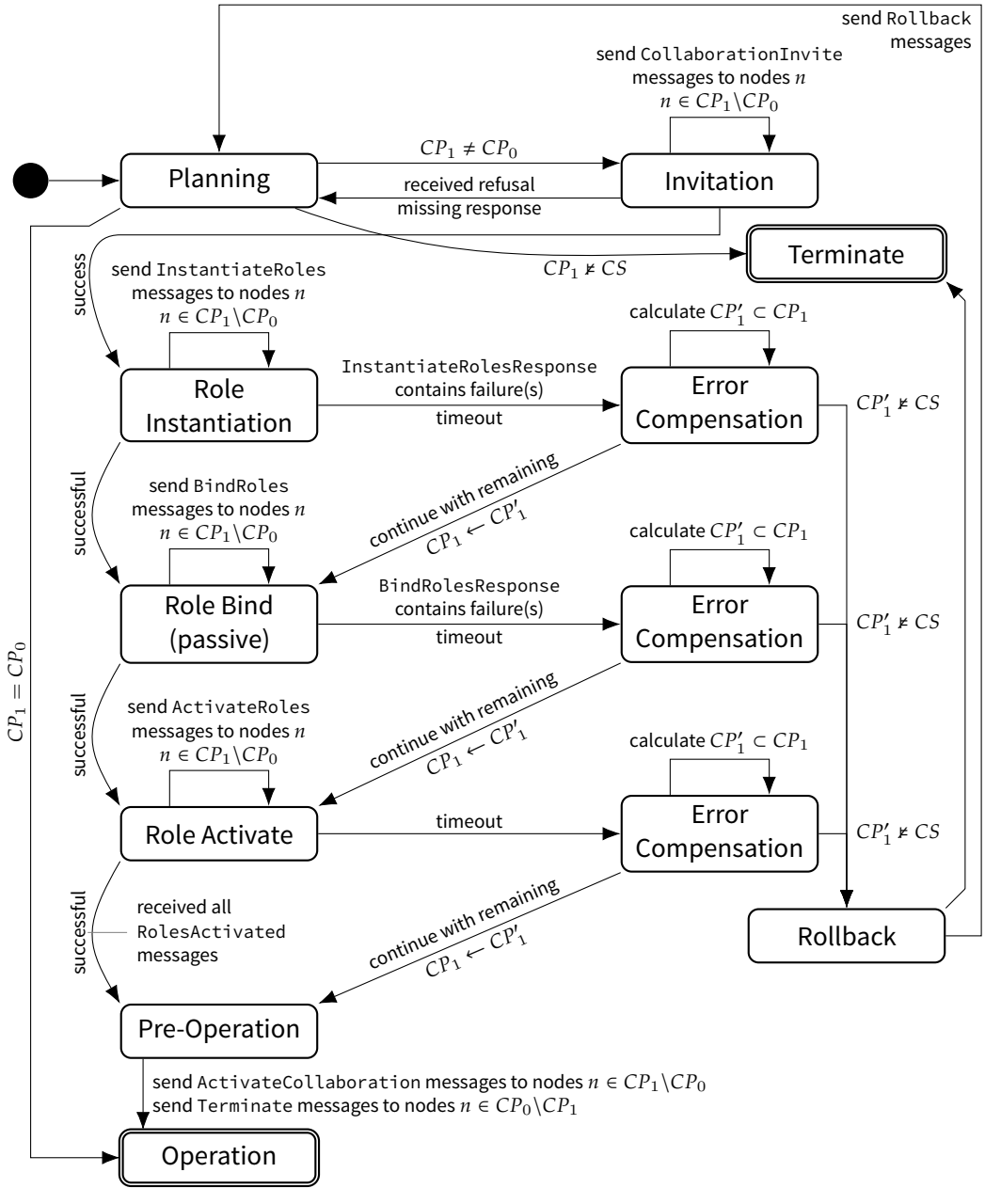


Figure 4.22: Complete View on the *Adaptation* State.

CP_1' cannot satisfy the collaboration specification ($CP_1' \nvDash CS$), it is uncertain that the infrastructure generally cannot satisfy the collaboration specification anymore. In contrast to the case mentioned afore, in which CP_1 did not satisfy the specification and therefore neither did CP_0, the initial plan created by the planner has been changed manually. The planner intentionally may have excluded some some roles that were originally part CP_0 so that $CP_0 \not\subset CP_1$. The subsystems providing those roles, however, may still be available. Additionally, it is uncertain that CP_0 is still valid with respect to the infrastructure. In order to ensure a consistent adaptation or to terminate the pervasive collaboration, the *Rollback* phase segues into another *Planning* phase, which results either in $CP_1 = CP_0$, which means that CP_0 is still valid and will be restored, $CP_1 \neq CP_0$, which means that another composition plan CP_1 could be computed resulting in another attempt to adapt, or $CP_1 \nvDash CS$, which means that no valid composition plan could be computed thus terminating the collaboration. Provided that adaptation proceeds successfully to the *Pre-Operation* state, `ActivateCollaboration` messages are sent to those systems that should add a new role, *i.e.*, $CP_1 \backslash CP_0$, and `Terminate` messages, containing the collaboration's and the respective roles' identifiers, are sent to those that should drop a role, *i.e.*, $CP_0 \backslash CP_1$. Since messages are assumed to be reliably transferred, no acknowledgement of these instructions is required. Consequently, the pervasive collaboration segues back into the *Operation* state, waiting for another adaptation or its proper termination.

4.3.6 Terminating a Pervasive Collaboration

The pervasive collaboration terminates either when the PCC leaves the collaboration or an adaptation cannot result in a valid composition plan, *i.e.*, $CP_1 \nvDash CS$. In both cases, the AS_{PCC} sends a `Terminate` message to all participating subsystems, which will then turn into a terminating state locally, as depicted in Figure 4.19 on page 73. The *Terminate* states on both the coordinating subsystem and the non-coordinating subsystems, locally clean up the dispatch tables, and thereby terminate the pervasive collaboration.

4.4 Summary

This chapter presented the RoleDiSCo approach in order to solve the problems stated in Section 1.3. Table 4.2 summarizes which aspect of the RoleDiSCo approach addresses which of the requirements posed in Section 1.4. The symbols denote a significant (■) or partial (⊡) contribution of the approach's aspect to the respective requirement.

The requirements posed by Smart Service Systems (RQ 1) were mainly addressed by the *Role-based Collaboration Specification*, which is a part of the *RoleDiSCo Development Methodology*. Though the main goal of the methodology was to demarcate the individual development processes (RQ 2), this demarcation significantly contributes to the feature of serendipity. Since both the *Collaboration Designer* and the *Phase ② Developer* do not have to take care of each other's work explicitly, serendipity is basically gained for free.

	Methodology	Specification	Middleware	Protocol
RQ 1 Smart Service Systems				
Complex Service Structures		■	■	■
Collaborative Nature		■	⊡	
Serendipity	■	■	■	
Context-Awareness		■	■	⊡
RQ 2 Demarcated Development Processes	■	⊡	⊡	
RQ 3 Run-Time Support for Smart Service Systems				
Automated Discovery		⊡	■	
Automated Composition & Adaptation		⊡	■	■
Decentralization & Infrastructure Abstraction			■	
RQ 4 Coordinated Composition and Adaptation of Smart Service Systems				
Complex Service Structures		(■)	■	■
Life Cycle States of Roles			⊡	■

Table 4.2: Review of the Requirements.

The discontinuity between design and run time has been addressed by the *RoleDiSCo Middleware*, which mainly addresses RQ 3, *i.e.,* the run-time support for Smart Service Systems. However, the middleware also contributes to the requirements of Smart Service Systems (RQ 1) in general. Thereby, the middleware especially has to realize the feature of serendipity at run time and has to manage the context-awareness. Nevertheless, it is responsible for deriving discovery information from the artifacts of Phase ① and to perform a decentralized discovery, which is the prerequisite for composition and adaptation. The middleware's infrastructure abstraction layer enables the middleware to operate in an arbitrary network infrastructure, especially in a decentralized environment.

In general, the requirements posed by Smart Service Systems have been addressed by utilizing the *Concept of Roles* in the RoleDiSCo approach. It inherently supports complex service structures and serendipity, thus, roles were not a requirement, but are a significant and beneficial part of the solution. Thereby they pose the requirement to deal with their run-time life cycle [46]. Consequently, the requirement RQ 4, *i.e.*, the coordinated composition and adaptation, was subdivided into the part concerning the complex service structures and the part concerning the run-time life cycle of roles. These requirements were addressed by the *Protocol for Coordinated Composition and Adaptation of Smart Service Systems*, which is a part of the middleware in order to achieve automated composition and adaptation of Smart Service Systems (RQ 3.2).

Conclusively, the contributions which are part of the *RoleDiSCo Approach* addressed the four key problems of engineering Smart Service Systems, all with the challenges of heterogeneity, context-awareness, and decentralization in mind: the missing specification for complex, context-aware, continuous service collaborations; the intertwined development processes; the discontinuity from design-time specification to run-time composition and adaptation; and on-demand composition of the desired systems.

5 Implementing RoleDiSCo

In order to evaluate the RoleDiSCo approach, the presented concepts were implemented in two research prototypes. The first comprises the *RoleDiSCo Development Methodology* and generates the partial implementation, which is then to be complemented manually. The resulting artifact can be used with the *RoleDiSCo Middleware*, which is the second prototype. Instead of discussing every single line of code, this chapter gives some important insights that are relevant to understand the evaluation presented in Chapter 6.

The code generation is specifically tailored to operate with the middleware although the generated source code might be usable for other purposes. The middleware, in turn, does not depend on the generated source code. Any existing code that adopts the super classes and annotations required by the middleware can be used, which allows to integrate the middleware with potential other role-based engineering approaches.

5.1 RoleDiSCo Development Support

A first step was to provide a role-based collaboration specification as the existing specifications, discussed in Section 3.1 on page 19, are not suitable for Smart Service Systems mainly due to their missing support for serendipity. Thus, a specification was developed by means of a DSL. A common and rather easy approach for developing DSLs as well as programming languages is the Xtext framework [7].

Xtext provides a powerful grammar language in order to define a DSL. The corresponding toolchain [86] allows to deliver a full development support, including parser, linker, typechecker, compiler as well as editing support for various development environments. Additionally, Xtext comes along with Xbase [26], which allows to easily incorporate programming language constructs into the Xtext grammar and provides a respective compiler generating Java source code out of the DSL. Xbase provides all utilities to easily create Java source code, but it is also possible to generate code in other languages. This, however, requires much more manual work since a complete compiler has to be

written for that language. Thus, the code generation will be limited to Java source code; the underlying concepts, however, are not limited to that language.

The DSL can be created either by manually defining its grammar or by generating the grammar from Ecore models [25] and manually adjusting it. Though the specification is based on a metamodel, the models of Xtext and Xbase would have to be integrated into that in order to receive a comprehensive model-based solution. In order not to interlink the different metamodels, the grammar for the role-based collaboration specification was defined manually, the complete version of which is listed in Listing B.2 on page 150. The first 109 lines define the entire structure of the collaboration specification whereas the remainder is solely required to interlink the specification's grammar with that of Xbase in order to detect the `player` references inside the method body and to prohibit the usage of `player` for other purposes.

At this point, it is already possible to generate a partial toolchain comprising an editor with syntax highlighting ans checking as well as partial type checking. The grammar is internally transformed to an ECore model. A concrete instance of the specification, as for example in Listing 6.1 on page 94, is parsed into an instance of such an ECore model. In order to generate the desired source code from the specification, the `IJvmModelInferrer` interface must be implemented. The implementation technically follows the process described in Section 4.1.2 on page 53: first, a collaboration class is generated with the respective annotations, as later shown in Listing 6.2 on page 96; next, one class is generated per role; finally, additional utility classes, such as the interfaces, are generated. Internally, this requires two processing rounds: initially, the types, methods, etc. are created in order to be able to reference them; then, the actual source code is generated. Otherwise, for instance, the specified role types cannot be used when creating the annotation within the collaboration class as these types would not exist yet.

Since Xbase is tailored to Java, it has a strong type system as well, which is challenging in with respect to serendipity. The specification should be able to define partial behavior by means of implementing methods which may delegate parts to the role's player, as described in Section 4.1.1 on page 51. The player should remain undefined in order to achieve serendipity and to decouple the development process. In order to reference a method within the specification, the player, however, must have a specific type that provides the method. Therefore, the player interfaces are generated, which do not have to be used by Phase ② developers, but allow to implement a type-safe code generation.

The complete code generation of the player interfaces is shown in Listing B.3 on page 152. Listing 5.1 shows the part thereof in which the player references are extracted from

Listing 5.1: Deriving the Player References of a Role's Method.

```
1 for(PlayerFeatureCall c : EcoreUtil2.getAllContentsOfType(feature?.body,
      PlayerFeatureCall)) {
2   val node2 = NodeModelUtils.findNodesForFeature(c,XbasePackage.Literals.
        XABSTRACT_FEATURE_CALL__FEATURE).head
3   val name = node2.text.trim
4   members += role.toMethod(name,if(c.explicitReturnType ) c.returnType.type.typeRef
        else Void.TYPE.typeRef) [
5     setDefault = false
6     setAbstract = true
7   ]
8 }
```

inside the role's method body. As mentioned, the specification is interlinked with the Xbase grammar. The `PlayerFeatureCall` represents a method call on the player and allows to traverse the model instance of the specification in order to search for these specific elements. Then, for each found method call, a method in the respective player's interface is created. Please note that each role has its own player interface.

Listing 5.2 on the next page shows how the role class' syntax tree is generated. The `accept` call registers the role as a class and thereby allows to reference the role type elsewhere. In Lines 4–11, a private player attribute, whose type is the interface generated in the preceding step, is injected into the role class. In the remainder, the role's features, comprising context, attributes, and methods are added as attributes and methods to the role class. Thanks to the usage of Xbase, the statement in Line 27 is sufficient in order to completely adopt the methods' structure. This implementation does not generate the source code but its syntax tree. The actual source code generation is achieved by the compiler, which is provided by Xbase. In order to reference the role types, *e.g.*, `Student.doSomething()`, they were added as attributes likewise.

After implementing the code generation process, using the model-driven approach for the generation of the grammar was realized to be a better option than the one that was chosen. The model-driven approach would allow to perform a model-to-model transformation before the actual syntax tree is computed in order to add a transient player attribute to each role type which will not be part of the syntax tree but only of its abstracting model, *i.e.*, the collaboration.

Listing 5.2: Deriving the Role's Implementation.

```
def dispatch void infer(Role role, IJvmDeclaredTypeAcceptor acceptor, boolean
    isPreIndexingPhase, CollSpec collSpec){
  acceptor.accept(role.toClass('''«collSpec.module.fullyQualifiedName».«role.
      name»Role''')) [
    documentation = role.documentation
    if(!isPreIndexingPhase) {
      superTypes += roleSuperType(role).typeRef
      var playerRef = typesFactory.createJvmField()
      playerRef.type = typeRef('''«collSpec.module.fullyQualifiedName».I«role.
          name»RolePlayer''')
      playerRef.simpleName = 'player'
      playerRef.visibility = JvmVisibility.PRIVATE
      members += playerRef
    }
    for(feature : role.features) {
      switch feature {

        Property : {
          members += feature.toField(feature.name, feature.type)
          members += feature.toGetter(feature.name, feature.type)
          members += feature.toSetter(feature.name, feature.type)
        }

        Operation : {
          members += feature.toMethod(feature.name, feature.type ?: Void.TYPE.typeRef
              ) [
            documentation = feature.documentation
            for(p : feature.params) {
              parameters += p.toParameter(p.name, p.parameterType)
            }
            body = feature.body
          ]
        }
      }
    }
  ]
}
```

5.2 RoleDiSCo Middleware

The RoleDiSCo middleware implements the basic features in order to operate in a decentralized environment as well as the coordinated composition and adaptation process. In this section, implementational details that have a certain impact on the evaluation in Chapter 6 are in focus. Either they contribute to the understanding of how the middleware achieves discovery, composition, and adaptation, or they affect the actual performance, discussed in Section 6.2 on page 113.

The middleware itself is implemented as a singleton class and is either started by a client, usually an application that also owns an artifact comprising a collaboration, its roles, and a complementing player implementation, or instantiated on its own. Once instantiated, the bootstrapping process will create a concrete infrastructure abstraction layer, establish the knowledge repositories and start the discovery process. Dependency injection [29] was utilized by means of Google Guice 4.1 [37], which allows to easily exchange parts of the middleware's implementation, even by third-party developers. In addition to that, Google Guava 22.0 [39] is used, which is a set of utility libraries that includes additional collection types, immutable collections, a graph library, and utilities for concurrency.

5.2.1 Infrastructure Abstraction Layer

An essential part of the middleware in order to operate in decentralized environments is an implementation of the infrastructure abstraction layer that actually copes with such environments. JGroups [5] is a library for reliable group messaging, written entirely in Java. It is based on IP multicast, and extended by *reliability* and *group membership*. Reliability in particular comprises that messages are retransmitted in case of message loss, fragmented messages are reassembled on the receiver's side, and the order of messages is preserved. Group membership includes that members in a group are aware of each other and are notified about changes. Consequently, JGroups (4.0.0) is used for the infrastructure abstraction layer, thereby avoiding any static centralized servers.

Additionally, JGroups allows to send arbitrary data objects as long as they implement Java's `Serializable` interface. Thus, this interface was implemented in the `Message` class, the super class of all message classes the modules of the middleware use for communication with other subsystems. Although this is a simple and quickly implementable approach, it is a rather slow solution at run time since Java's `Serializable` interface

transforms the object into a byte array of its memory representation. Implementing a customized serialization mechanism could result in lower encoding and decoding times.

JGroups' documentation warns the developer of the issue of time-consuming message decoding, which is why messages are decoded in a separate thread in order not to block the inbound communication channel. Figure 5.1 shows a scatter plot of the message

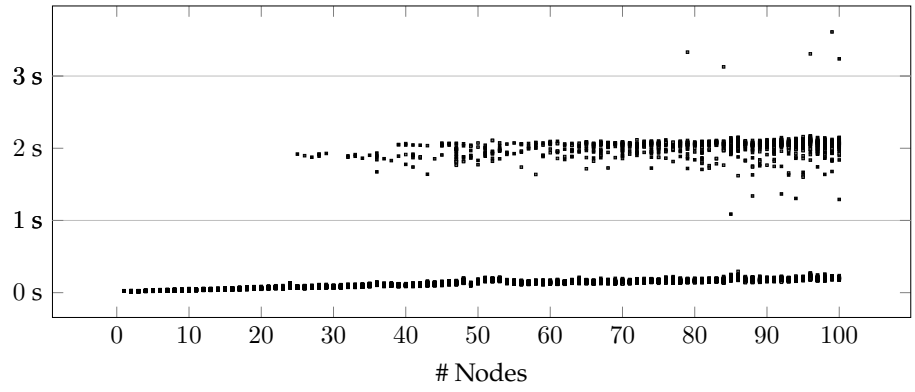


Figure 5.1: Message Processing Time.

processing time measured in one of the very early experiments evaluating scalability. Therein, the time a node required to obtain and process all the discovery information from other nodes was measured. In the scenario used in this particular experiment, the 100th node, for instance, had to process 198 messages as two `RoleAnnouncement` messages were sent from each other node. The experiment was repeated 100 times, thus, 10,000 data points were collected, each representing how much time was required to process all the messages. The marks below 1 s represent the behavior as it was originally expected, *i.e.,* messages were processed basically instantaneously. However, starting with 25 nodes, data points indicate a nearly constant processing time of 2 s with a few outliers above 3 s at 80 to 100 nodes. The huge gap between the clusters indicates that the reason for this behavior cannot be solely due to limited computing resources. The reason for this issue is that JGroups silently drops messages if the number of threads cannot satisfy the amount of incoming messages. Other mechanisms ensure that a message is sent a second or even a third time which explains the huge, empty gap between the clusters. Consequently, the implementation of JGroups was slightly adopted with respect to the implementation of the RoleDiSCo middleware, *i.e.,* the thread pool used by JGroups was exchanged with an implementation that does not drop incoming messages. This may slightly increase the overall average processing time, but the results in general are much better since messages do not have to be resent due to a lack of available threads.

5.2.2 Knowledge Repositories and Local Class Discovery

The knowledge repositories, such as those for discovery information or collaboration specifications, are implemented as in-memory collections. The *directory service*, which technically holds data obtained by the discovery module, uses Google Guava's [39] `Multimaps`, similar to a key-value storage except that values are again collections. Depending on the concrete request this can be a very fast or very slow data structure, which has to be kept in mind for evaluating the performance. The collaboration's type was chosen the as the key, storing all the discovery information related to that type within the respective collection. However, updating this data structure is already quite challenging due to concurrency and the different types of changes, *e.g.,* add, remove or update. Keeping the data in memory is still the fastest viable option.

In order to populate the repository containing the collaboration specifications, the generated classes were annotated during the code generation process. These annotations are processed using the *ClassIndex* framework [67], which basically preprocesses the annotations during compile time and creates an index which is static at run time. The `AbstractCollaboration` class was annotated with `@IndexSubclassed` and the `@Collaboration` annotation was annotated with `@IndexAnnotated`. Thereby, the collaboration classes can be easily indexed as shown in Listing 6.8 on page 104.

5.2.3 Planner

In order to compose a collaboration, a composition plan needs to be computed first. Computing such a plan based on a collaboration's specification and the discovery information results in a constraint satisfaction problem, the solving of which is a research field of its own. Planners for such problems were assumed to exist.

OptaPlanner [72], for instance, is a constraint satisfaction solver including a lightweight, embeddable planning engine. It allows to solve optimization problems efficiently, as constraints can be applied on the level of plain domain objects and thereby reuse existing code. The composition plan, hence, could be computed based on a composition plan's metamodel or derived from an adjusted code generation process providing a specification tailored for OptaPlanner. Although OptaPlanner was initially investigated for this purpose, this approach was not further pursued as the main focus of this thesis is to achieve automated composition and not to provide an optimized composition plan.

Consequently, a rather simple planning solution tailored to the collaboration's structure was implemented. Therefore, the `kind` attribute was added to the generated collaboration's class annotations in order to easily determine the right planner at run time. Since the structure is defined at design time, computing the collaboration's structure can be done during design time as well, thereby avoiding the necessity for doing such computation at run time. As a general-purpose planner was initially assumed, this property was not introduced in the conceptual part. The basic planner, implemented as part of the research prototype, provides a greedy solution which takes all provided discovery information and converts it into composition instructions as long as the role's types and contexts match those of the collaboration.

6 Evaluation

One of the main research goals of this thesis is to investigate the applicability of the *Concept of Roles* for on-demand composition of Smart Service Systems. Smart Service Systems, operating in smart computing environments, are a subset of distributed systems, which aim to unite the decentralized and emerging features of SOSSs with the complex, predefined structures of other systems. They face three major challenges, *i.e.*, *heterogeneity*, *context-awareness*, and *decentralization*. In order to overcome the key problems of engineering such systems, the RoleDiSCo Development Methodology and the RoleDiSCo Middleware were proposed. This chapter evaluates the proposed solutions against the problem statements and requirements, explained in Sections 1.3 and 1.4.

Composing several independent services, the development lanes of which are demarcated, is challenging, especially if the collaboration itself is another independent development lane. Not to impose any dependencies was an important goal, which is achieved by RoleDiSCo Development Methodology. It provided the missing specification for complex, context-aware, continuous service collaborations and demarcated the previously intertwined development processes in one go. A case study, presented in Section 6.1, is used in order to demonstrate how the development processes were separated and how this development methodology paves the way for on-demand composition, thereby building the foundation to bridge the gap between design and run time. The RoleDiSCo Middleware complements the development methodology in order to completely eliminate the discontinuity between design and run time. In Section 6.2, the middleware, including the protocol for coordinated composition and adaptation, is evaluated based on the implemented research prototype, in order to provide some metrics concerning the overall performance, scalability, and overhead.

The case study and the provided insights into the performance and scalability of the middleware show that role-based composition solves the problems of on-demand composition of Smart Service Systems in decentralized environments. Lastly, the *Concept of Roles* is taken up in Section 6.3 once more in order to discuss benefits and drawbacks with respect to composition of Smart Service Systems.

6.1 Case Study: Distributed Slideshow

The first part of the evaluation focuses on the functional aspects of the proposed solutions. The scenario below is used in order to stepwise explain how an application is described using the proposed specification, how the generated partial implementation looks like, how this implementation is to be complemented by Phase ② developers, and, eventually, how this results in on-demand composition and subsequent adaptation.

6.1.1 Scenario

The Classroom scenario introduced in Chapter 4 is a typical scenario in the domain of role-based modeling. Though it is easy to motivate that a *Student* is played by a *Person* and that there are multiple instances of a person who may play this role, it is, however, hard to motivate that the *Student* role is played by different autonomous services, thereby the need for flexibility becomes obsolete. Instead, a slightly different example is used, which follows a similar structure, *i.e.*, a one-to-many structure, but acts in a different setting having a need for more flexibility: a distributed, on-demand slideshow, depicted in Figure 6.1. Consider you are in a bar and want to share photos with your friends. Instead handing around the smart phone, which invites other people to snoop for photos they should not see, the photos should be displayed on devices having the Viewer role. You, the Presenter, initiate the decentralized slideshow, *i.e.*, you activate an app, select pictures and basically create a session on demand. Smart devices having the Viewer role can join and leave the collaboration over time. All devices are

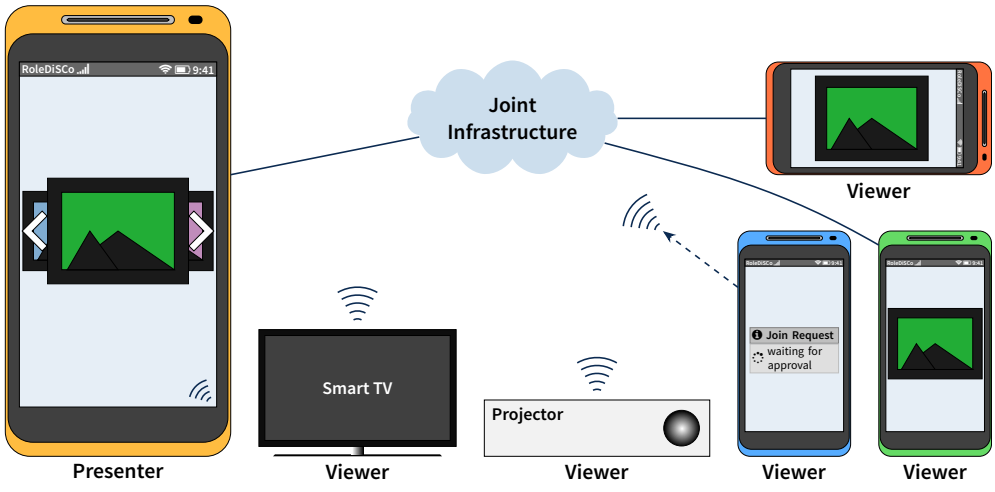


Figure 6.1: Scenario of the Distributed Slideshow.

assumed to be attached to a joint network infrastructure. Between the Presenter role and its corresponding Viewer roles, a one-to-many relationship exists. The environment is decentralized and volatile as devices can (dis-)appear over time. Communication is asynchronous and bidirectional as Viewers are able to give feedback to the Presenter. Additionally, in a different environment, such as a smart home, other devices such as a smart TV or even a legacy projector are conceivable to be incorporated.

In this scenario, it is more evident to motivate Presenter and Viewer as roles, as their actual execution remains unclear at design time of the collaboration, *i.e.,* it is not clear whether the role will be played by a smart phone, a tv set, a legacy projector or even on a device which has to preprocess pictures in order to provide them to their users. Moreover, it is easy to imagine that, for instance, the default photos application of a mobile operating system provides plugin infrastructures, which enables third-party developers to extend or use that application, and in this case enables Phase ② developers to attach the Presenter or Viewer roles to the default photos application. The case study was not implemented on a real mobile platform, but as a generic Java application instead as JGroups, the infrastructure abstraction layer of the RoleDiSCo middleware, is a Java implementation and hence would require the Android [2] mobile platform but JGroups is not tailored for Android. Moreover, Android is not a role-based mobile platform and yet there is no role-based mobile platform available in general. Conclusively, there was no benefit of developing a mobile application for Android compared to a pure Java one.

6.1.2 Phase ①: Collaboration Design

In Phase ①, the *Collaboration Designer*, specifies the distributed slideshow using the collaboration specification, as depicted in Listing 6.1 on the following page and discussed below. Thereinafter, the derived partial implementation will be explained in detail in order to show how it will contribute to on-demand composition eventually.

Scenario Specification

The declaration in Line 1 sets the namespace of the entire collaboration, thereby, enclosing the collaboration and its roles. The fully qualified collaboration type results from the namespace and the name given in Line 2: `org.rosi.roledisco.samples.ds.DistributedSlideshowCollaboration`. The fully qualified role types are generated likewise, *i.e.,* `org.rosi.roledisco.samples.ds.PresenterRole` and `org.rosi.roledisco.samples.ds.ViewerRole`. Both the collaboration and the Viewer role have

Listing 6.1: Collaboration Specification of the Distributed Slideshow Scenario.

```
1  module org.rosi.roledisco.samples.ds
2  collaboration DistributedSlideshow {
3    context Session session
4    coordinator role Presenter {
5      player op receiveFeedback(String message)
6      op setPicture(Picture picture) {
7        val pictureBase64 = Utils.convertToBase64(picture)
8        Viewer.setPicture(pictureBase64)
9      }
10   }
11   role Viewer {
12     context Session session
13     op submitFeedback(String message) {
14       Presenter.receiveFeedback(message)
15     }
16     op setPicture(String pictureBase64) {
17       val picture = Utils.convertFromBase64(pictureBase64)
18       player.setPicture(picture)
19     }
20   }
21   multiplicities {
22     Presenter one-to-many Viewer
23   }
24   constraints {
25     Presenter >-< Viewer
26   }
27 }
```

a context attribute of type `Session`, named `session`, which refers to a common class in the designated target language, which is Java as the prototypes were implemented in Java. `Session`, hence, is a Java class, the source code of which is shown in Listing B.4 on page 153. This class is assumed to exist as a data structure and should be manually provided by the collaboration designer. For the sake of simplicity, it holds a session identifier, thereby representing a unique instance of a Distributed Slideshow collaboration at run time, so that others can join this particular session. Throughout the remainder of this section, a simple string is used as identifier, which can be considered as some kind of pre-shared key. In a real-world scenario, however, this context should be derived from friendship relations, location, etc.

Next, starting with Line 4, the Presenter role is specified. The `receiveFeedback` method is declared to be fully delegated to a concrete player at run time, while the `setPicture` method contains a partial implementation, which converts the given image to a Base64-encoded string. This behavior is solely relevant to the collaboration but not to the player. Subsequently, the encoded image string is passed to the `setPicture` method

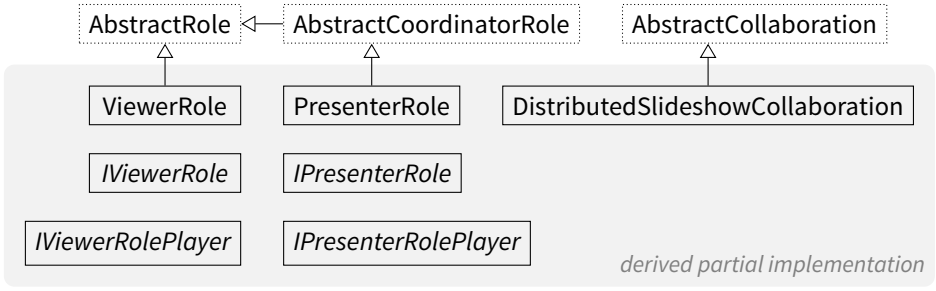


Figure 6.2: Generated Partial Implementation for the Distributed Slideshow Scenario.

of the Viewer roles. Multiple Viewer roles are considered since Line 22 specifies a *one-to-many* relationship between the Presenter role and the Viewer roles. Additionally, this relationship is constrained, so that a Presenter role must not be located on the same device as a Viewer role for a concrete instance of a collaboration.Lines 11 to 20 specify the non-coordinating Viewer role. The `submitFeedback` method is a simple outbound method call, which invokes the `receiveFeedback` method on the Presenter role. The Viewer role's `setPicture` method reverses the encoding before the `setPicture` method is invoked on the player, which is determined at run time.

For the sake of completeness, the `Picture` class is a customized wrapper of Java's native `File` class and `Utils` is a utility class which encodes and decodes pictures to Base64 and vice versa. Both classes are created by the collaboration designer and are intended to be part of the bundle which is also containing the derived partial implementation.

Derived Partial Implementation

This specification results in a partial implementation, shown in Figure 6.2. It comprises one class for the collaboration, one class per role, and a set of utility classes, which are intended to support the Phase ② developer but are not required for later composition.

The generated `DistributedSlideshowCollaboration` class, shown in Listing 6.2 on the following page, contains the structure of the collaboration and will be used at run time in order to recreate the collaboration specification. Therefore, a set of annotations, *i.e.*, `@Collaboration`, `@Constraint`, and `@Multiplicity`, was generated, which additionally enables the middleware to find the annotated classes automatically. Please note that `@Collaboration#kind` is a manually added property, explained in Section 5.2.3, in order to specify the concrete planner already at design time. Here, the middleware is instructed to use a planner for simple one-to-many collaborations.

Listing 6.2: Excerpt of the Generated Collaboration Class. (cf. Listing B.5 on page 154)

```
1  package org.rosi.roledisco.samples.ds;
2  /* ... */
3  @Collaboration(
4    coordinator = PresenterRole.class, roles = ViewerRole.class,
5    kind = Collaboration.Kind.ONE_TO_MANY )
6  @Constraint(
7    from = PresenterRole.class, to = ViewerRole.class,
8    value = RoleConstraint.ROLE_PROHIBITION )
9  @Multiplicity(
10   from = PresenterRole.class, to = ViewerRole.class,
11   value = RoleLink.ONE_TO_MANY )
12 public class DistributedSlideshowCollaboration extends AbstractCollaboration {
13   @Context Session session;
14
15   public DistributedSlideshowCollaboration(Session session) {
16     super();
17     this.session = session;
18     CompositionManagement.compose(this);
19   }
20   /* ... */
21 }
```

Line 13 refers to the `Session` context attribute, which is later utilized by the middleware in order to match context properties as some kind of membership restrictions. The constructor listed in Lines 15 to 19 sets the appropriate context when the collaboration is instantiated and triggers the composition by calling the middleware's composition management. The context (`session`) is required to be passed to the constructor in order to use it throughout the composition and thereby to select appropriate participants (role types) for the collaboration. The coordinating role may be passed as another parameter to the constructor (see Listing B.5), which then would be forwarded in conjunction with the instantiated collaboration to the composition management.

Listing 6.3 shows the generated implementation of the non-coordinating Viewer role. The code of the coordinating Presenter role is generated likewise, except for the context property, as shown in Listing B.6 on page 155. The `@Context` annotation in Line 4 notifies the middleware's context manager to search for appropriate context providers. Since roles in RoleDiSCo are not instantiated as long as they are not participating in a collaboration, they are not able to provide dynamic context as they only exist as types.

The subsequent method implementations are the respective generated methods of the role methods in Listing 6.1 on page 94. Depending on the multiplicity of the target role, a different call to the dispatcher is generated (see Listing B.6), which reduces the need of

Listing 6.3: Generated Viewer Role.

```
1 package org.rosi.roledisco.samples.ds;
2 /* ... */
3 public class ViewerRole extends AbstractNonCoordinatorRole {
4
5   @Context Session session;
6
7   public void sendFeedback(String message) {
8     Dispatcher.getDispatcher().dispatchToRole(this, PresenterRole.class, "
          receiveFeedback", msg);
9   }
10
11  public void setPicture(String pictureBase64) {
12    Picture picture = Utils.convertFromBase64(pictureBase64);
13    Dispatcher.getDispatcher().dispatchToPlayer(this, "setPicture", picture);
14  }
15 }
```

continuously evaluating the model at run time in order to decide whether one or several instances have to be addressed. The dispatcher receives the current role instance (`this`), the target role type, the method name and its parameters. Since a concrete role instance always exclusively belongs to one concrete collaboration instance, that collaboration instance can be determined automatically. The concrete collaboration is required in order to find the appropriate corresponding role instance. Since the dispatcher also acts as an interface between the middleware and a potential local role-based runtime, the call to a concrete player instance is delegated to the dispatcher first.

6.1.3 Phase ②: Player Complementation

In Phase ②, potentially several developers complement the artifacts generated up to this point, which includes to provide the desired performance of the role as well as the corresponding context. Provided that all applications would entirely operate on role-based runtimes, the role could simply be bound to the existing service, which then acts as the player for that role. In reality, however, a huge variety of platforms and runtimes exists which is why it needs more than simply binding the role to the service. Thus, different approaches to use the derived partial implementation together with a (potentially previously) existing service are discussed first in order to show that the RoleDiSCo approach works with both role-based runtimes and non-role-based run-time environments. Thereafter, the context provisioning is explained.

Providing the Concrete Performance

The two perspectives on the concept of roles are reflected in role-based runtimes to some extent. For instance, *LyRT* [76] is a recent role-based runtime and is rather player-centric than organization-centric. It provides a *Dynamic Instance Binding* mechanism in order to bind role instances to core objects (*i.e.,* players). Though *LyRT* supports the notion of compartments, they do not necessarily form the strong shape of an objectified collaboration as it would be required for an organization-centric approach. *LyRT* dynamically resolves method calls from players to its roles and vice versa. Role instances are bound programmatically to player instances. This is a very flexible approach as players can be arbitrary objects, *i.e.,* any player may play any role at run time, or more precisely, *LyRT* as described in [76] does not even distinguish whether a given object is actually a player, a role or a compartment. *LyRT* also supports loading new role types at run time as well as reloading existing role types at run time without affecting existing instances. [MW7]

The role types generated as part of the RoleDiSCo methodology are self-contained classes, similar to the role types in *LyRT*. The generated collaboration class can be used as compartment in *LyRT*. Thereby it is possible to instantiate and distinguish multiple collaboration instances and the respective roles inside. Since players in *LyRT* can be arbitrary objects, there are no special considerations except that the player must provide the required methods. Otherwise, method calls will result in a run-time exception.

By contrast, *ObjectTeams* (OT/J) [43] is an organization-centric runtime and relies on an implicit binding based on a *played-by* relation, *i.e.,* all player instances of a certain type that *can* play a role, *will* play that role (when their encapsulating team is active). For the sake of completeness, it should be mentioned that a more fine-grained control of binding roles can be achieved using *guard predicates* [44], which is a language construct of OT/J. In OT/J, *Teams*, similar to collaborations in RoleDiSCo, found the building blocks of the language. Roles are specified as parts of teams. However, they also remain part of their team at run time, *i.e.,* they are not translated to independent classes in between, which does not match the generated code provide by the research prototype described in Section 5.1. This would require a different approach to create code processable by OT/J. Apart from that, providing a complementing player implementation can be achieved as follows: assuming a `DistributedSlideshowCollaboration` team class containing the `PresenterRole` and `ViewerRole` was generated (Listing 6.4:2–5), a Phase ② developer has to subclass the team class and replace the `playedby` relation with a concrete player class as shown in Lines 8–11 of Listing 6.4 on the next page.

Listing 6.4: Complementing Player Implementation exemplified in OT/J.

```
1  // derived partial implementation
2  public team class DistributedSlideshowCollaboration {
3    public class Presenter {/* ... */}
4    public class Viewer {/* ... */}
5  }
6
7  // complementing implementation
8  public team class AComplementedDistributedSlideshowCollaboration extends
       DistributedSlideshowCollaboration {
9    public class Presenter playedBy APresenterPlayer {/*...*/}
10   /*...*/
11 }
12
13 // player class, i.e., original service implementation
14 public class APresenterPlayer {/* ... */}
```

Please note that all the classes shown in Listing 6.4 actually reside in separate class files. At this point, the demarcation of the individual development lanes becomes clearly visible: Lines 2–5 are the generated collaboration, Line 14 is the complementing player which may have existed earlier, and Lines 8–11 show the actual task a Phase ② developer needs to do: bind the role to its player. In case of OT/J, the strong typing system may limit the role-player interaction on the level of the generated collaboration to some extent.

Systems that do not run on top of a role-based runtime and hence cannot simply bind a role, are denoted as legacy systems. Consider, for instance, the plain old projector that should become able to join a collaboration. Assuming the projector's internal system to be able to run arbitrary applications, the runtime could not be replaced with a role-based one. Additionally, this can be used for services or applications to be integrated that cannot be shut down or changed in order to operate on top of a role-based runtime. In this case, the complementing *player* implementation acts as an adapter between the original implementation and the role, shown in Figure 6.3. Listing 6.5 illustrates how

Figure 6.3: Legacy Player Implementation.

such a complementing implementation could be done in Java. The complementing player class may use the player's interface definition, which is useful in this particular

Listing 6.5: Legacy Player Implementation in Java.

```
class OldProjectorViewer /* implements IViewerRolePlayer */ {
  private void setPicture(Image picture) {
    ProjectorRuntime.getRuntime().setBackgroundImage(picture);
  }
}
```

case, as it defines all the methods that have to be provided. Please note that the player class does not extend any classes that are part of the metamodel or the middleware.

In fact, the case study hereinafter relies on the legacy option by means of the `DistributedSlideshow` class, shown in Listing 6.6. Schütze and Castrillon [71] pointed out that various role-based runtimes perform very differently at run time due to different dispatching mechanisms. This, however, has only small effects on the composition and adaptation aspects this thesis is concerned with as this only triggers the instantiation of a role. The `DistributedSlideshow` class, whose complete implementation is shown in Listing B.7 on page 155, acts as a player for both the Presenter and Viewer role.

Listing 6.6: Excerpt of the `DistributedSlideshow` Class as Legacy Player.

```
package org.rosi.roledisco.samples.ds.ui;
/* import statements */
public class DistributedSlideshow {
  private List<Picture> pictures;
  private int imageIndex = 0;
  private ViewerRole viewerRole;
  private PresenterRole presenterRole;
  /* ... */

  /* Constructor for the Presenter Role. Instantiated manually. */
  public DistributedSlideshow(String sessionName, List<Picture> pictures) {
    PresenterPlayerProvider.getInstance().addPresenter(sessionName, this);
    new DistributedSlideshowCollaboration(new Session(sessionName));
    this.pictures = pictures;
    image.setIcon(new ImageIcon(pictures.get(0).getAbsolutePath()));
    nextButton.addActionListener(e -> {
      int newImageIndex = (imageIndex + 1) % pictures.size();
      if (presenterRole != null) {
        presenterRole.setPicture(pictures.get(newImageIndex));
      }
      DistributedSlideshow.this.setPicture(pictures.get(newImageIndex));
      imageIndex = newImageIndex;
    });
    previousButton.addActionListener(e -> {
      /* ... */
    });
  }
```

```
28
29   /* Constructor for the Viewer Role. Instantiated via ViewerPlayerProvider */
30   public DistributedSlideshow(String sessionName) {
31     /* ... */
32     feedback.addActionListener(e -> {
33       if (viewerRole != null) {
34         viewerRole.sendFeedback(feedback.getText());
35         feedback.setText("");
36       }
37     });
38     /* ... */
39   }
40
41   private void receiveFeedback(String message) {
42     comments.append(String.format("%s\n", message));
43   }
44
45   private void setPicture(Picture picture) {
46     image.setIcon(new ImageIcon(picture.getAbsolutePath()));
47   }
48
49   /* ... user interface setup ... */
50 }
```

Providing Player and Context to the Middleware

Another task of the Phase ② developer is to provide the appropriate context information for a player. Therefore, the *Context Provider* interface was introduced, which has to be implemented for this purpose. Additionally, the middleware needs to be able to choose the appropriate player for the provided context, which is achieved by implementing the *Player Provider*. Listing 6.7 is a combined implementation of both interfaces for the `DistributedSlideshow` player class and the Viewer role. The corresponding implementation for the Presenter role is shown in Listing B.8 on page 157.

Listing 6.7: Combined Context & Player Provider for the Viewer Role.

```
1  package org.rosi.roledisco.samples.ds;
2  /* import statements */
3  public class ViewerPlayerProvider implements ContextProvider, PlayerProvider {
4    private static final ViewerPlayerProvider myInstance = new ViewerPlayerProvider();
5
6    static {
7      Dispatcher.getDispatcher().addPlayerProvider(ViewerRole.class, getInstance());
8      ContextManager.getContextManager().addContextProvider(ViewerRole.class,
           getInstance());
9    }
10
11   private final Vector<String> sessions = new Vector<>();
```

```
12   private final Vector<Context> contexts = new Vector<>();
13
14   public static ViewerPlayerProvider getInstance() {
15     return myInstance;
16   }
17
18   @Override
19   public Collection<Context> getContext(Class<? extends AbstractRole> role) {
20     return contexts;
21   }
22
23   @Override
24   public boolean hasPlayer(Class<? extends AbstractRole> roleType, Context context) {
25     Session session = (Session) context.getValueForKey(new ContextFeature("session"))
           .getValue();
26     for (String s : sessions) {
27       if (s.equals(session.getSessionId())) return true;
28     }
29     return false;
30   }
31
32   @Override
33   public Object getPlayer(Class<? extends AbstractRole> roleType, Context context) {
34     Session session = (Session) context.getValueForKey(new ContextFeature("session"))
           .getValue();
35     if (sessions.contains(session.getSessionId())) {
36       return new DistributedSlideshow(session.getSessionId());
37     }
38     return null;
39   }
40
41   public void addSession(String session) {
42     sessions.add(session);
43     contexts.add(new Context(new ContextFeature("session"), new ContextValue(new
           Session(session))));
44   }
45 }
```

Line 21 provides the required context information which is computed through the `addSession` method in line 41. For the sake of generalization, the `Session` class is not used directly, but wrapped inside a generic model, in which `Context` is a set of feature-value pairs, as already illustrated in the metamodel in Figure 4.2. It is in evidence that more specific context providers must become part of the code generation process in order to provide better support to Phase ② developers. As of now, for instance, it is hard to enforce the Phase ② developers to provide all the required context information except checking for completeness at run time.

Next, the middleware needs to be able to decide which player to instantiate for the case that several options were available. Here, only one `ViewerPlayerProvider` class exists,

but there may be more than just this one. Nevertheless, the `hasPlayer` method returns true in case a session identifier was added earlier. Session identifiers are added via the main application, shown in Listing B.9 on page 158. In case the requested session identifier is in the local list of session identifiers, the `getPlayer` method instantiates the player with the appropriate session identifier as constructor parameter and returns that player instance. Choosing the constructor wisely enables the `DistributedSlideshow` player class to provide implementations for both role types, as shown in Listing 6.6. Please note that the role type already has to be passed when the interface implementations are registered to their respective managers, hence, the methods' *role type* parameter has to be verified only in case multiple role types are served with the same provider implementation.

With respect to the legacy projector, a crucial issue at this point is that the legacy projector might not be able to provide dynamic context information. Considering the session identifier not to be a dynamic property but rather a pre-shared key, it has to be changed for each new collaboration, such as by typing it in via a remote control. A simple solution would be to modify the player provider in a way that it returns always `true` for any given context. Then, the planner on the collaboration initiator's device could decide whether or not to invite the legacy projector in a collaboration, even if context information is missing. This, however, should require further, rather generic, information about the systems that want to join a collaboration since the planner would be asked to invite a systems it does not know anything about. For the sake of simplicity, compositions are created based on simple context matching, which however is not a strict limitation of the approach itself. Improvements of this process are part of future work.

6.1.4 Coordinated Composition and Adaptation at Run Time

Up to this point, the development phases, *i.e.*, Phase ① and Phase ②, were discussed. The remainder of this section addresses the run-time aspects and highlights how the derived partial implementation in conjunction with the Phase ② developers' adjustments contribute to automated discovery, composition and adaptation.

System	Role & Responsibility
AS_1	Presenter role, thus AS_{PCC}
AS_2	Viewer role, part of initial composition, leaves after AS_4 joins
AS_3	Viewer role, part of initial composition, but binding fails (forcedly)
AS_4	Viewer role, part of adaptation, joins before and leaves after AS_2 leaves

Table 6.1: Systems, their Roles, and the Situation they showcase.

Next, the Distributed Slideshow scenario is extended by (simulated) devices hereinafter. Similar to the conceptual part, AS_n refers to a device. Since the scenario will establish a one-to-many collaboration, the number of devices to be considered can be limited to four, as shown in Table 6.1, in order to showcase the different situations that can occur. For the sake of clarity, symbolic names are used instead of UUIDs [59] to address the individual subsystems. Please note that a concrete complementing player implementation is usually unknown to remote subsystems. Consequently, no further details concerning their implementation are available, except that respective players are assumed to exist and to provide the respective context features if necessary. In this case study, however, the complementing player, *i.e.*, the `DistributedSlideshow` class, is used on all subsystems, but this information is private to each local subsystem and not shared.

Discovery

Before a collaboration can be composed, AS_1 to AS_4 need to discover each other, their respective roles and contexts. During the middleware's bootstrapping phase, the generated collaboration and role classes are preprocessed, as described in Section 5.2.2 on page 89, in order to recreate the structure at run time as shown in Listing 6.8. Subsequently, they are added to the repository containing the collaboration specifications.

Listing 6.8: Indexing the Generated Collaboration and Role Classes.

```
1  private void updateCollaborationSpecifications() {
2    for (Class<? extends AbstractCollaboration> clazz : ClassIndex.getSubclasses(
         AbstractCollaboration.class)) {
3      if (!collaborationSpecifications.containsKey(clazz)) {
4        if (clazz.isAnnotationPresent(Collaboration.class)) {
5          addCollaboration(clazz,
6                           clazz.getAnnotation(Collaboration.class).coordinator(),
7                           clazz.getAnnotation(Collaboration.class).roles());
8        }
9      }
10   }
11 }
```

The `addCollaboration` method triggers the discovery module, whose implementation is shown in Listing B.10 on page 159, to broadcast a `RoleAnnouncement` message to the infrastructure. The role types, *i.e.*, `ViewerRole` and `PresenterRole`, are checked whether they are actually discoverable (Listing B.10, Lines 66–83). Since the Presenter role does not have any context, the dispatcher module is checked, whether a complementing player exists for that role directly. In case of the Viewer role, the context is

requested from the context manager and the dispatcher is asked whether players exist for the respective contexts. Lastly, the `RoleAnnouncement` messages are created with the contents shown in Listing 6.9 for the Presenter and Viewer role, respectively.

Listing 6.9: `RoleAnnouncement` Messages for the Presenter & Viewer Roles.

```
1  RoleAnnouncementMessage (PresenterRole):
2    Subsystem: AS-1
3    CollaborationType: org.rosi.roledisco.samples.ds.DistributedSlideshowCollaboration
4    RoleType: org.rosi.roledisco.samples.ds.PresenterRole
5
6  RoleAnnouncementMessage (ViewerRole):
7    Subsystem: AS-1
8    CollaborationType: org.rosi.roledisco.samples.ds.DistributedSlideshowCollaboration
9    RoleType: org.rosi.roledisco.samples.ds.ViewerRole
10   Context: Session(1234)
```

Upon receiving the `RoleAnnouncement` message, *e.g.,* on AS_2, the remote discovery module processes this message and adds the contained information to the directory service (cf. Listing B.10, Lines 101–108). Since AS_1 has been newly discovered on AS_2 and the respective message was sent as a broadcast instead of a directed message, which can be inferred from the receiver's address being empty, the discovery module on AS_2 processes this event in order to reply with AS_2's role announcements as described afore. AS_1, however, will not reply with a role announcement then as AS_1 received a directed message, thereby assuming that AS_2 already knows of AS_1.

Composition

At this point, AS_1, AS_2, and AS_3 discovered each other in order to be able to collaborate. How to achieve automated, coordinated composition is explained hereinafter.

Initially, the `DistributedSlideshow` player class (Listing 6.6) instantiates the generated `DistributedSlideshowCollaboration` class (Line 13). Prior to that the player instance registered itself to the Presenter role's player provider (Listing B.8) with the corresponding session identifier. The generated collaboration class subsequently invokes the *composition management* upon instantiation (Listing 6.2, Line 18), which in turn triggers the composition process, as depicted in Listing 6.10, Line 7.

Listing 6.10: Composition Management – Initiating a Collaboration.

```
1 public static void compose(AbstractCollaboration collaboration) {
2   RoleDiscoMiddleware.getCompositionManagement().doCompose(collaboration, null);
3 }
```

```
4 private void doCompose(AbstractCollaboration collaboration, AbstractCoordinatorRole
      coordinatorRole) throws CompositionFailedException {
5   CoordinatingPervasiveCollaboration pervasiveCollaboration =
        pervasiveCollaborationFactory.create(collaboration, coordinatorRole);
6   collaborationMap.put(pervasiveCollaboration.getCollaborationID(),
        pervasiveCollaboration);
7   pervasiveCollaboration.compose();
8 }
```

The actual composition process is shown in Listing 6.11. First, the specification is retrieved in Lines 2–9 from the repository containing the collaboration specifications. Next, the dispatcher is requested whether the coordinating role has an appropriate player (Lines 11–15). If necessary, the coordinating role is instantiated (Lines 15–21) before it is bound to its newly instantiated player (Line 24). Therefore, the context is extracted from the `DistributedSlideshowCollaboration` instance with the help of the `@Context` annotated attributes. In this concrete example, the collaboration was instantiated without the coordinating role instance, which is why the `PresenterRole` will be instantiated at this point. An instance of the `DistributedSlideshow` player class is already known as a player to the middleware thanks to the Presenter role's player provider. Consequently, this player, which also instantiated the collaboration and triggered the composition process, will be selected as the coordinating role's player. After that, the collaboration segues into the *Planning* state.

Listing 6.11: Pervasive Collaboration Coordinator (PCC) – Initialization.

```
1 private void initialize() {
2   Optional<CollaborationSpecification> specification = collaborationManager.
        getCollaborationSpecifications().stream().filter(spec -> spec.
        getTheCollaboration().equals(collaboration.getClass())).findFirst();
3
4   if (!specification.isPresent()) {
5     logger.error("Collaboration␣{}␣does␣not␣have␣a␣corresponding␣specification!",
          collaboration.getClass().getSimpleName());
6     throw new RuntimeException("Missing␣specification!");
7   }
8
9   CoordinatingPervasiveCollaboration.this.specification = specification.get();
10
11  Class<? extends AbstractCoordinatorRole> coordinatingRoleClass = specification.get
        ().getTheCoordinatorRole();
12  if (!dispatcher.hasPlayer(coordinatingRoleClass)) {
13    logger.error("Failure!␣No␣complementing␣player␣available!");
14    throw new MissingPlayerException(specification.get());
15  }
16
17  this.context = extractContext();
18
19  if (coordinatorRole == null) {
```

```
20      coordinatorRole = dispatcher.createCoordinator(coordinatingRoleClass);
21      if (coordinatorRole == null) {
22        logger.error("Failure! Coordinating Role could not be instantiated!");
23        return;
24      }
25    }
26
27    Context tmpContext = coordinatorRole.getContext() == null ? this.context :
          coordinatorRole.getContext();
28    Object coordinatorPlayer = dispatcher.getPlayer(coordinatorRole, tmpContext);
29    if (!dispatcher.bind(coordinatorRole, coordinatorPlayer)) {
30      logger.error("Failure! Player for coordinating role could not be bound!");
31    }
32    setState(CompositionState.PLANNING);
33  }
```

Therefore, a concrete planner is assigned for the entire lifetime of the collaboration, as shown in Listing 6.12. The *planner factory* provides a planner specific to the collaboration's structure. As mentioned with respect to Listing 6.2 on page 96, the concrete planner is determined based on the `kind` property of the `@Collaboration` annotation. In this concrete scenario, the `SimpleOneToManyPlanner`, which is the greedy planner described in Section 5.2.3, is returned and used.

Listing 6.12: Pervasive Collaboration Coordinator (PCC) – Planning.

```
1  private void plan() {
2    if (planner == null) planner = PlannerFactory.create(specification);
3    compositionPlan = planner.calculateCompositionPlan(directoryService.
         getDiscoveryKnowledge(specification));
4    if (compositionPlan == null || compositionPlan.isEmpty()) {
5      setState(CompositionState.TERMINATE);
6    } else {
7      setState(CompositionState.PCC_SEND_INVITES);
8    }
9    this.compose();
10 }
```

The planner is instantiated with the given collaboration specification. Then, the composition plan is calculated invoking the `calculateCompositionPlan` method and passing the discovery information, which is exmplified in Table 6.2. Finally, the planner pro-

System	Role Type	Context
AS_2	`ViewerRole`	Session(1234)
AS_3	`ViewerRole`	Session(1234)

Table 6.2: Content of the Directory Service on AS_1 (simplified).

vides a composition plan transforming every piece of information into a composition instruction, such as instantiating the Viewer role with context *Session(1234)* on AS_2.

Listing 6.13: Preprocessing and Sending of `CollaborationInvite` Messages.

```
1 private void sendInvites() {
2   Multimap<AdaptiveSubsystem, CompositionInstruction> instructionMultimap =
        getCompositionInstructionsPerSubsystem();
3   for (AdaptiveSubsystem adaptiveSubsystem : instructionMultimap.keySet()) {
4     inviteMessages.add(CollaborationInviteMessage.create(specification.
          getTheCollaboration(), adaptiveSubsystem, collaborationID,
          instructionMultimap.get(adaptiveSubsystem)));
5   }
6   inviteTimer = initializeTimer(inviteTimer, CompositionState.PCC_WAIT_FOR_INVITES,
        this::forceFinishInvitationState);
7   setState(CompositionState.PCC_WAIT_FOR_INVITES);
8   infrastructureAbstractionLayer.send(inviteMessages);
9 }
```

Next, the `CollaborationInvite` messages are created and sent to AS_2 and AS_3. Listing 6.13 shows how the composition instructions are preprocessed in order to send only one invite to each participating subsystem. The `Multimap` data structure is a key-value data structure for which *value* is a collection of the specified type. Here, each subsystem is related to a collection of composition instructions. After preparing the actual messages, a timer is set to handle timeouts (Line 6). Right before the messages are sent, the collaboration segues into the waiting state. Apart from different contents of the messages created and send hereinafter, preprocessing works similar for all of them.

Listing 6.14: `CollaborationInvite` Message sent to AS_2.

```
1 CollaborationInviteMessage:
2   Subsystem: AS-2
3   Collaboration ID: f94f1772-0896-4bf8-8639-a4feb52dee30
4   Role: org.rosi.roledisco.samples.ds.ViewerRole
5   Context: Session(1234)
```

The `CollaborationInvite` message shown in Listing 6.14 is processed by the composition management of AS_2 and a similar message by that of AS_3. Thereby, a local representation of the collaboration to be composed is instantiated in order to distinguish several collaborations and participate in those. Listing 6.15 shows the message processing. In Lines 3–10, the actual provisions are calculated, thereby ensuring that the local system has a player for the given role in the given context. With respect to the scenario, only one role type in one context is to be instantiated, thus the provisions can be completely

assured, so that both AS_2 and AS_3 return an `InviteAcknowledgement` message to AS_1. For the case that not a single provision can be made, an `InviteRefused` message would be sent to AS_1 and the local collaboration would terminate (cf. Listing 6.15, Line 12).

Listing 6.15: Collaboration Management on AS_2 and AS_3 – Initialization.

```
1  public void processMessage(CollaborationInviteMessage message) {
2    setState(CompositionState.REMOTE_INITIALIZATION);
3    List<Class<? extends AbstractNonCoordinatorRole>> roleTypes = message.getRoleTypes
         ();
4    List<Context> contexts = message.getContexts();
5    for (int i = roleTypes.size() - 1; i >= 0; i--) {
6      if (!dispatcher.hasPlayer(roleTypes.get(i), contexts.get(i))) {
7        roleTypes.remove(i);
8        contexts.remove(i);
9      }
10   }
11   if (roleTypes.isEmpty()) {
12     infrastructureAbstractionLayer.send(InviteRefusedMessage.create(message));
13     this.terminate();
14   } else {
15     infrastructureAbstractionLayer.send(InviteAcknowledgementMessage.createResponse(
           message, roleTypes, contexts));
16   }
17 }
```

AS_1 receives the two `InviteAcknowledgement` messages, which are processed as shown in Listing 6.16. In this case, the composition plan does not need to be updated because

Listing 6.16: Processing `InviteAcknowledgement` Messages on AS_1.

```
1 private void processMessage(InviteAcknowledgementMessage message) {
2   if (states.peek() == CompositionState.ADAPTATION) {
3     currentAdaptation.processMessage(message);
4   } else if (states.peek() == CompositionState.PCC_WAIT_FOR_INVITES) {
5     inviteAcknowledgmentMessages.add(message);
6     compositionPlan.update(message.getSource(), message.getRoleTypes(), message.
          getContexts());
7     this.checkForStateCompletionAndContinue(CompositionState.PCC_WAIT_FOR_INVITES,
          inviteMessages, inviteAcknowledgmentMessages, inviteRefusedMessages,
          inviteTimer);
8   } else { /* ... */ }
9 }
```

all provisions were assured. When all messages are processed (cf. Listing B.11), the collaboration segues into the *Role Instantiation* state, sending `InstantiateRoles` messages to AS_2 and AS_3, the processing of which is shown in Listing 6.17. Both will successfully

instantiate the role and return an `InstantiateRolesResponse` message containing the unique identifiers of the concrete role instances for the given types and contexts to AS_1.

Listing 6.17: Processing `InstantiateRoles` Messages on AS_2 and AS_3.

```
1 public void processMessage(InstantiateRolesMessage message) {
2   setState(CompositionState.ROLE_INSTANTIATION);
3   InstantiateRolesResponseMessage.Builder responseBuilder =
        InstantiateRolesResponseMessage.Builder.create(message);
4   for (int i = 0; i < message.getRoleTypes().size(); i++) {
5     AbstractNonCoordinatorRole role = dispatcher.createRole(
6       message.getRoleTypes().get(i), message.getContexts().get(i));
7     if (role != null) {
8       localRoles.put(role.getId(), role);
9       responseBuilder.addBoundRole(role);
10    } else {
11      responseBuilder.addBindFailure(message.getRoleTypes().get(i), null);
12      this.rollback();
13    }
14  }
15  infrastructureAbstractionLayer.send(responseBuilder.build());
16 }
```

Next, AS_1 sends a `BindRoles` message to both AS_2 and AS_3. The middleware's dispatcher module, cf. Section 4.2.5, instantiates the player prior to the actual binding operation (Listing 6.18, Line 4) with the help of the `ViewerPlayerProvider` (Listing 6.7), which actually instantiates the player. Two cases are to be distinguished at this point: the binding operation is assumed to be successful on AS_2 but to fail on AS_3 as, for example, the player could not be instantiated. Consequently, both AS_2 and AS_3 respond with a

Listing 6.18: Processing `BindRole` Messages on AS_2 and AS_3.

```
1 public void processMessage(BindRolesMessage message) {
2   setState(CompositionState.ROLE_BINDING);
3   BindRolesResponseMessage.Builder builder = BindRolesResponseMessage.Builder.create(
        message);
4   for (int i = 0; i < message.getRoleIDs().length; i++) {
5     AbstractNonCoordinatorRole role = localRoles.get(message.getRoleIDs()[i]);
6     if (dispatcher.bind(role, dispatcher.getPlayer(role, role.getContext()))) {
7       builder.addBoundRole(role.getId());
8     } else {
9       builder.addBindFailure(role.getId(), null);
10    }
11  }
12  infrastructureAbstractionLayer.send(builder.build());
13 }
```

`BindRolesResponse` message but with different contents. AS_2 will add the successfully instantiated role's identifier to the set of bound roles, while AS_3 will add the respective identifier to the set of failures. AS_1 receives both messages and has to update the composition plan accordingly by removing the composition instruction for AS_3 and deleting the role id for the Viewer role on AS_3. Additionally, the updated composition plan has to be validated against the collaboration specification. In this case, the updated composition plan is still valid since the Presenter role on AS_1 and the Viewer role on AS_2 are available.

Next, the `ActivateRoles` message is sent to AS_2, which then activates the role and responds with a `RolesActivated` message. Since no other systems are included in the plan anymore, AS_1 can immediately send an `ActivateCollaboration` message, which includes binding information relevant to the receiving subsystem. In this particular case, it is solely the Presenter role on AS_1, which is included in this message. At this point, the collaboration on AS_1 and its local representation on AS_2 segue into the operational state.

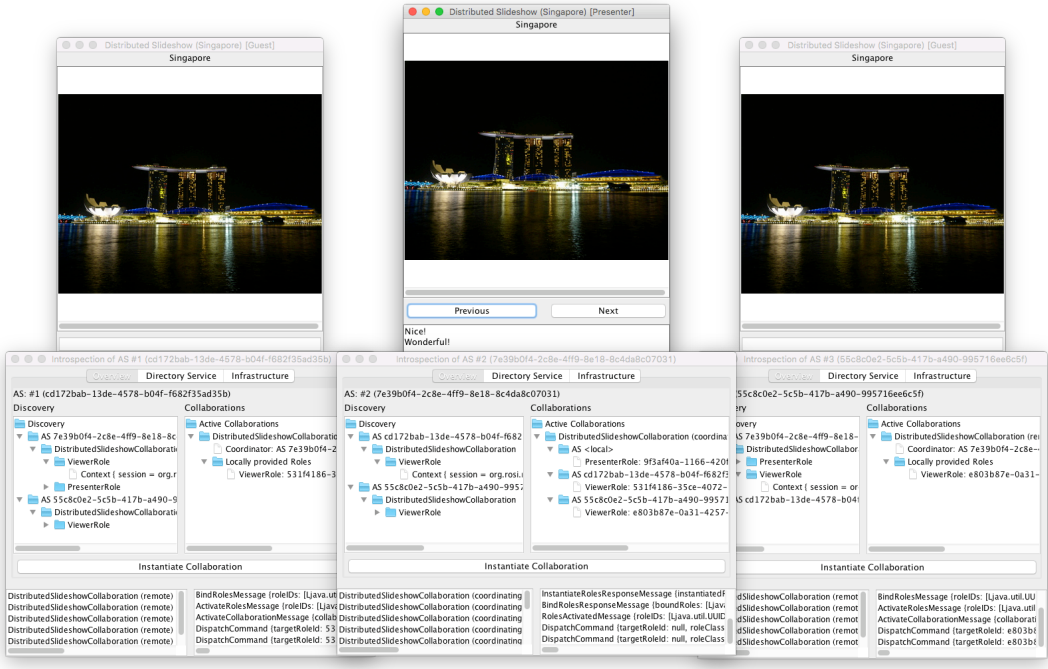


Figure 6.4: Operational Distributed Slideshow Collaboration.

Figure 6.4 shows an operational Distributed Slideshow collaboration with three subsystems. The host is placed in the center of the figure. Aside, two guests will see the photo the host is currently watching. Also, they can send some comments which only the host will see. Below the application, an introspection window presents some run-time information of the middleware, such as discovery knowledge or the collaboration structure.

Adaptation

Adaptation is considered a recomposition of an operational collaboration. Since the adaptation process is similar to the composition process, as explained in Section 4.3.5 on page 76, solely the differences compared to the composition process are discussed.

First, an adaptation which integrates a new system, *i.e.*, AS_4, is considered. AS_4 joins the infrastructure and the discovery process is starting as described earlier. A respective event triggers the composition management on AS_{PCC} to recalculate the plan based on the newly acquired discovery information. It comprises newly added information relevant to the current collaboration, in this particular case that a new Viewer role is available on AS_4 in context *Session(1234)*.

CP_0

AS	Role Type	Context
AS_1	`PresenterRole`	-
AS_2	`ViewerRole`	Session(1234)

CP_1

AS	Role Type	Context
AS_1	`PresenterRole`	-
AS_2	`ViewerRole`	Session(1234)
AS_4	`ViewerRole`	Session(1234)

Changeset ADD ($CP_1 \backslash CP_0$)

AS	Role Type	Context
AS_4	`ViewerRole`	Session(1234)

Changeset REMOVE ($CP_0 \backslash CP_1$)

AS	Role Type	Role ID
	$\emptyset$	

Figure 6.5: Composition Plans CP_0 and CP_1 and Derived Change Sets.

Therefore, the collaboration segues into the *adapatation* state and therein into the (adaptation's) *planning* state, in which the composition plan is updated according to the newly available discovery information. Figure 6.5 shows the original composition plan CP_0, its updated version CP_1 and the derived change sets. Obviously, the Viewer role has to be instantiated in the appropriate context on AS_4. At this point, the new role is integrated the very same way as it would be done in the initial composition, described afore. Please note that the roles to be instantiated (or terminated) are only taken from the given change sets and not from the actual composition plan. In case of any failures, the entry is removed from both the composition plan (CP_1) and the change set, and the whole plan is checked against the collaboration specification.

Next, an adaptation is considered, in which the context of the Viewer role provided by AS_2 changes to *Session(4321)*, causing the planner to exclude this role from the collaboration in an updated plan, shown in Figure 6.6. The Viewer role on AS_2 will be terminated by sending a `Terminate` message, comprising the collaboration's and the role's identifier.

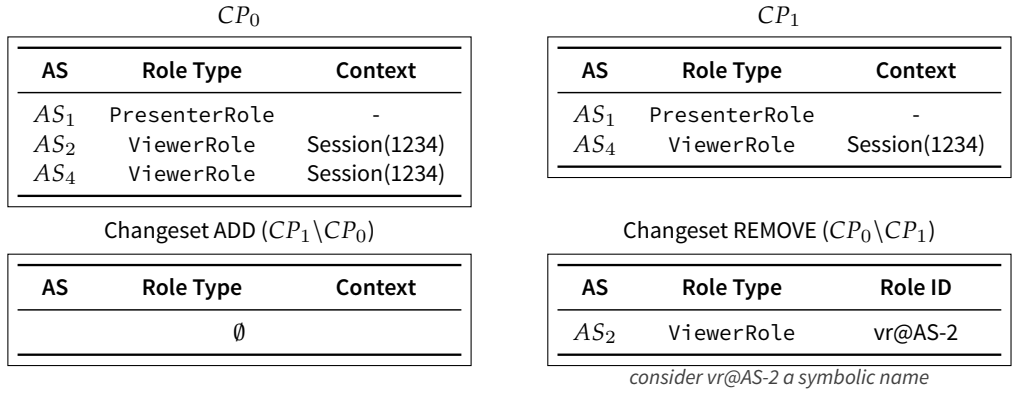


Figure 6.6: Composition Plans CP_0 and CP_1 and Derived Change Sets.

Eventually, AS_4 leaves the infrastructure, so that only the Presenter role remains. Again, the planner calculates an updated plan. In this case, however, the updated plan will be empty because the *one-to-many* relationship cannot be fulfilled any longer. As the composition plan now is empty, the collaboration is torn down rigorously.

6.2 Runtime Evaluation

Time is a crucial issue for composing continuous service collaborations in decentralized environments. In user-centric application scenarios, such as the distributed slideshow or the tech-enhanced classroom, a long initial composition phase would prevent users from using the application. Nielsen [62] has identified three important thresholds: 100 ms lets the users feel that the system reacts instantaneously; 1 s is the limit in which the users feel that they are able to freely navigate and their flow of thought continues to stay uninterrupted, even though they will notice the delay; and 10 s is the delay for keeping the user's focus on the application. In the domain of connected cars and connected cities, time is even more critical as delays may lead to accidents or cause injury.

Consequently, the proposed middleware including the composition protocol, is evaluated in terms of time with respect to different system sizes and different application scenarios. The prototypical implementation of the RoleDiSCo Middleware will be used to conduct the evaluation in an emulative way in order to obtain concrete performance results. Since discovery is a prerequisite for composition, the performance of the discovery mechanism is evaluated first, and that of the composition protocol subsequently. Prior to that, the general testbed setup and the scenarios used for the evaluation are ex-

plained. How the experiments for discovery and composition were performed and how respective data was obtained is explained in the respective section of each experiment.

6.2.1 General Testbed Setup and Scenarios

In order to keep the experiments controllable and to reduce any effects caused by the physical network, *e.g.,* delay or unexpected message loss, the testbed was placed on a single physical machine. That means, a *node* is actually one instance of the scenario application launched on the physical machine. JGroups is used as *Infrastructure Abstraction Layer*, and configured to use UDP multicast. Network communication was restricted to the loopback interface. All experiments were conducted on two identical physical machines, which were desktop PCs with an Intel Core i5-4590 CPU of 3.30GHz, and 16 GB memory. Debian GNU/Linux 9.1 (stretch) with Linux Kernel 4.9 was used as operating system. The Java Virtual Machine used for the experiments was an OpenJDK 1.8.0-141.

Due to the variety of role-based runtimes, RoleDiSCo's fallback implementation of the *Dispatcher* module was used instead of a specific role-based runtime. The fallback implementation mocks a role-based runtime to a certain degree. Since the focus of this thesis is on discovery and composition, this is a valid simplification since significant performance impacts occur on the level of method dispatch and not during role instantiation or binding, according to Schütze and Castrillon [71].

Scalability with respect to the composition protocol is affected by the system size, *i.e.,* the number of nodes, and the number of roles contained in the respective collaboration specification. Thus, two rather abstract scenarios, which capture different application structures, were created in order to obtain an insight how these structures affect the composition process and thereby to evaluate the middleware and the composition protocol with respect to different kinds of applications. The first scenario, denoted as Simple-Sample, is derived from the use cases mentioned. It represents a one-to-many collaboration, which consists of one coordinating role and one non-coordinating role, as well as respective players. The second scenario, denoted as Five-Roles, is rather artificial and intended to gain an insight how the composition process behaves if each node provides more than one role instance. It consists again of one coordinating role type, but five non-coordinating role types. The coordinating role type holds a one-to-many relationship to each non-coordinating one. In order to automate and streamline the experiments' execution, a wrapping class for both scenarios was used, which instantiates the middleware and conditionally idles or triggers the composition process after 10 s, which is enough time to complete the discovery process.

6.2.2 Discovery Time

The discovery is responsible for sharing information within and gathering such from the infrastructure, and thereby it is the prerequisite for composition and adaptation. Hence, discovery time is an initial delay the system needs to wait before joining or initiating a collaboration. Besides, the performance of the discovery process acts as a metric for the quality of the testbed environment itself. Thereby, the required time for discovering other nodes, is considered a baseline with respect to scalability of the general system behavior within the testbed in order to approve the testbed's setup to be appropriate.

Discovery usually relies on additional heartbeat messages in order to detect lost and new subsystems. As the whole testbed is placed on one physical machine, the experiment solely relies on the `RoleAnnouncement` messages in order not to bias the results. This works as follows: a new node broadcasts discovery information, including locally available collaboration types, role types and context information. Upon receiving such a broadcast message, it is clear that the originating node exists in the infrastructure and the receivers reply directly with a similar message containing their local information.

Discovery time, here, is defined as the time between broadcasting the first message and processing all the responses of other systems available in the infrastructure. In a decentralized environment, however, a local runtime cannot determine how many *other* systems should be available in order to know when discovery is complete. Thus, the experiment setup takes over this responsibility. For each application scenario, a series of 30 experiments was conducted, thereby gradually increasing the number of involved nodes from 5 to 150 in steps of 5 nodes. This equates to 300 messages to be processed for the Simple-Sample scenario and to 900 messages for the Five-Roles scenario as one message is sent per node and role type. Thus, the Five-Roles scenario is expected to require thrice as much time than the Simple-Sample scenario to complete the discovery process.

In order to approve that all the discovery information had been processed, the middleware was configured to create log files, which were analyzed during the experiment's execution. The discovery time was then measured on the last node instantiated per experiment. Each experiment was repeated 510 times on both machines. The last instantiated node was recreated for every measurement, which amounts to 510 times on each machine, in order to repeat the discovery process. The results of the first ten iterations were dropped for each experiment on both machines in order to avoid results being biased by initial bootstrapping of the nodes to be discovered.

This resulted in an initial data set of 1000 values per experiment, of which all values larger than 2 s were considered as outliers. Though this seems arbitrary, it is related to limited system resources which causes a retransmission of messages, which introduces a delay of at least 1.5 s, as explained in Section 5.2.1 with respect to Figure 5.1. A complete explanation of this issue is provided in Appendix A, which also includes the reasons why it is valid to exclude all values larger than 2 s. Thereof, the mean $\overline{r}$ and the standard deviation σ were calculated and only results within $[\overline{r} - 2\sigma, \overline{r} + 2\sigma]$ were kept.

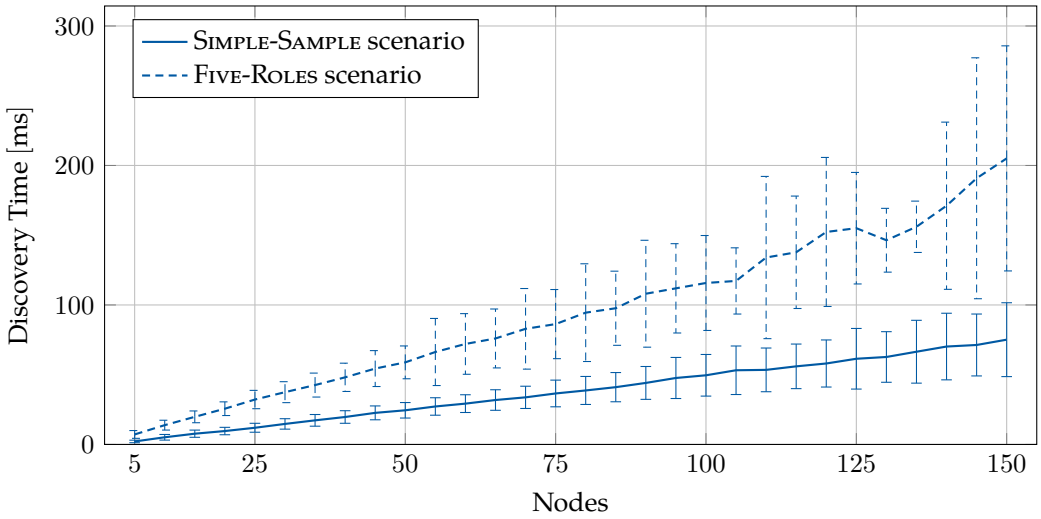


Figure 6.7: Discovery Time of # Nodes in ms.

Figure 6.7 shows the results for both scenarios: both lines depict the average discovery time the respective node required to process all the discovery information. Overall the testbed performs well for both scenarios. The results for the Five-Roles scenario are more volatile due to the adjusted, and thereby reduced, data set especially for larger system sizes, as depicted in Figure A.2 on page 146. Additionally, the Five-Roles scenario requires approximately thrice the time of the Simple-Sample scenario to complete the discovery process thereby meeting the expectations posed earlier. The results of the Simple-Sample scenario indicate that the testbed works reliable for up to 150 nodes or 300 messages to be sent and received rather simultaneously. Though those of the Five-Roles scenario seem to provide a similar statement for up to 900 messages and 150 nodes, the extended discussion in Appendix A on page 145 reveals that sending and receiving 300 messages rather simultaneously is the upper boundary for which the testbed and the prototypical implementation provides reliable results.

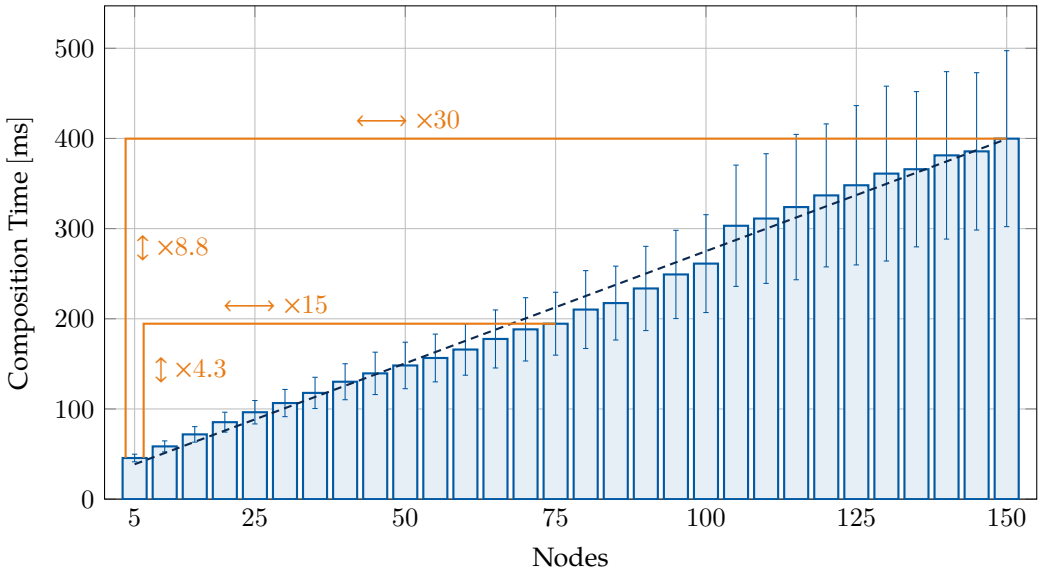
6.2.3 Composition Time

The previous evaluation indicates that the testbed scales reasonably well and that the results for up to 300 messages sent and received rather simultaneously should be reliable. Next, the time required to compose the scenario applications among a set of several nodes in order to establish a collaboration is to be investigated. This *composition time* is considered the overhead the composition process causes compared to static or predefined composition.

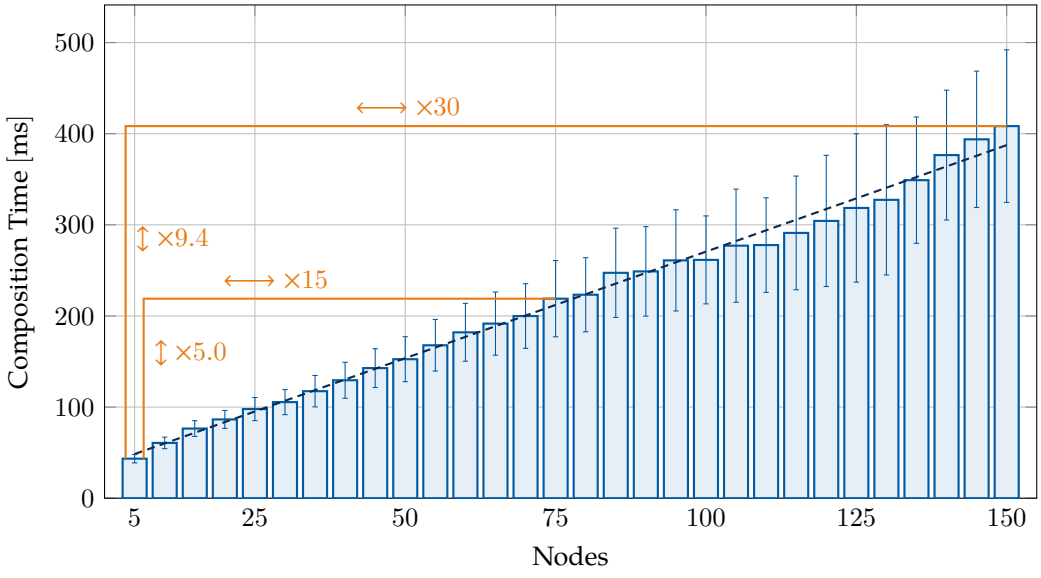
More precisely, *composition time*, in this thesis, is defined as the time between initiating a collaboration and reaching its operational state, depicted in Figure 4.10 on page 66. Thus, it also covers the planning phase. This allows to include a minimal planning time within the results that acts as a baseline for other potential planner implementations, which are considered to consume more time, as any further sophisticated planning may depend on a concrete use case, thereby increasing the overall composition time. For that purpose, the greedy planner, presented in Section 5.2.3, was used, which basically converts all entries in the discovery knowledge into a composition instruction, *i.e.*, every node that provides the non-coordinating role type will be instructed to instantiate that.

Similar to the experiments for the discovery time, for each application scenario, a series of 30 experiments was conducted. In each experiment, the number of involved nodes was increased by 5, reaching from 5 to 150 nodes. The last node instantiated per experiment gained the position of the PCC and thereby triggered the composition process. Each experiment was repeated 510 times on both machines. The node representing the PCC was recreated for every composition, so 510 times on each machine. The results of the first ten iterations were dropped for each experiment on both machines in order to avoid results being biased by initial bootstrapping of the non-coordinating nodes. This resulted in a data set of 1000 values per experiment, of which the mean $\overline{r}$ and the standard deviation σ were calculated and only results within $[\overline{r} - 2\sigma, \overline{r} + 2\sigma]$ were kept in order to remove severe outliers.

Provided that the PCC has to send five messages and process four messages per collaborating system independent of the actual scenario, the system is expected to scale linearly to the number of nodes. Since for the Five-Roles scenario the fivefold number of roles of the Simple-Sample scenario has to be instantiated, its average composition time is expected to be greater or equal to that of the Simple-Sample scenario. As the role-based runtime is simply mocked using the middleware's fallback implementation, it is possible that the average composition time for both scenarios is almost equal since *bind* and *activate* operations do not require complex computation and return almost immediately.



(a) Composition Time for the Simple-Sample Scenario.



(b) Composition Time for the Five-Roles Scenario.

Figure 6.8: Average Composition Time per System Size and Scenario.

Figure 6.8 depicts the average composition time correlated to the number of nodes involved, based on the adjusted data for both experiments. Both experiments in general scale reasonably well. Concerning the Simple-Sample scenario, cf. Figure 6.8a, the system scales very well up to approximately 100 nodes. Starting with 105 nodes the system still scales well, but a small increase is noticeable and results become more volatile. This is a consequence of placing the testbed on one physical machine, which slowly runs out of resources at that point. Though outliers were systematically removed, the impact of limited system resources is still in evidence. It is noteworthy that increasing the number of nodes by a factor of 15 only increases the composition time by a factor of 4.3. Even for a set of 150 nodes, which is a scenario 30 times larger than the smallest one, the composition process takes only 8.8 times longer. Overall, 150 nodes were composed in 399.8 ms which is less than the threshold of 1 s response time for interactive systems. [62]

Concerning the Five-Roles scenario, depicted in Figure 6.8b, the system scales continuously well up to approximately 80 nodes. Starting with 85 nodes the system still scales well, but results become volatile too. The reason why 100 nodes are allegedly composed as fast as 95 nodes is that more outliers were excluded for 100 nodes, which results in a smaller, thus better, data set and causes a lower average composition time. This holds true for 90 and 110 nodes, respectively. Here, increasing the number of nodes by a factor of 15 increases the composition time by a factor of 5, and for a set of 150 nodes, composition time increases by a factor of 9.4. Overall, 150 nodes were composed in 408.3 ms.

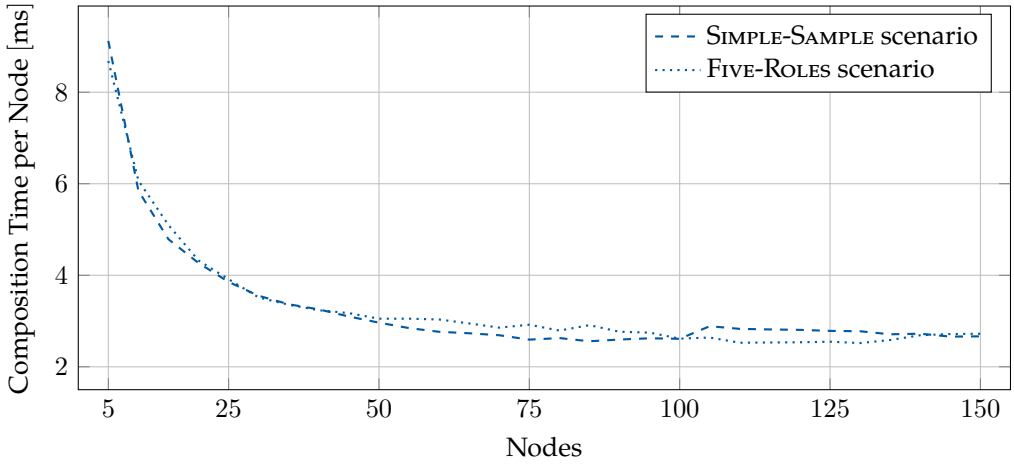


Figure 6.9: Composition Time per Node.

Figure 6.9 depicts the ratio of the total average composition time to the number of nodes involved. Interestingly, this ratio is better the larger the system size gets. Since intentionally the total composition time and not just the time which is required for

message processing was used, it is evident that the actual computation for planning or message serialization is more significant for rather small collaborations than for larger ones. In this concrete setting, it is recognizable that there will be a lower bound of approximately 2.5 ms/node. Especially for the Simple-Sample scenario, the effect of limited system resources is observable as well: a small increase starting at around 105 nodes. The reason why the actually more computing-intensive Five-Roles scenario behaves better than the Simple-Sample is discussed in the next section.

6.2.4 Discussion

First, scalability and performance of the composition process itself are subject to further discussion. Intentionally, the actual workload was kept rather small in order to evaluate the performance and scalability of the composition process itself rather than those of the applications. The evaluation showed that the RoleDiSCo approach scales linearly to the number of nodes, while the number of roles to be instantiated on a single node is of rather limited significance. Thereinafter, other aspects of the performance and scalability evaluation, such as the relation to other scenarios, the potential impact of a role-based runtime, and how adaptation is addressed, are qualitatively discussed.

For both scenarios, Figure 6.9 indicates that the composition time eventually scales linearly to the number of nodes, which coincides with the expectations, although an overhead for small system sizes is in evidence. As shown in Figure 6.8, the composition time amounts to approximately 400 ms for both scenarios. Though it was not a requirement, staying below the mark of 1 s response time for interactive systems [62] is a great achievement in order to establish this solution in real-world scenarios and applications as the user's train of thought is not interrupted. Evidently, the composition time in real-world scenarios will heavily depend on the actual application structure and use case. However, the user's patience is assumed to correlate with the application's complexity and system size to some extent – which of course applies only to user-centric applications.

Comparing the average composition time for 80 nodes of both scenarios, the Five-Roles scenario is 6.2 % slower than the Simple-Sample scenario, which corresponds with the expectations as well. Figure A.3 on page 147 provides a complete comparison of both scenarios' composition times and their relation. Nevertheless, this situation is inverted starting with 100 nodes. This anomaly, *i.e.*, the fact that the less computing-intensive Simple-Sample scenario allegedly composes slower than the Five-Roles scenario, is a result of limited system resources, especially of limited threads.

Figure A.4 on page 147 visualizes the partial composition time of each phase for 45 to 100 nodes based on a subset of the previous data set. Evidently, the waiting phases for `BindRolesResponse` and `RolesActivated` messages take much longer for the Simple-Sample scenario than for the Five-Roles scenario. Responses are only accepted in the respective waiting state, which is why the middleware's implementation switches to that state just before sending the messages as otherwise a response could be received while sending messages is still in progress. Thus, the time for the waiting state actually comprises the sending, remote processing, and receiving of messages. Due to the minimal workload of the *bind* and *activate* operations, the remote systems in the Simple-Sample scenario reply faster than those in the Five-Roles scenario, thereby potentially hindering the process of sending messages. Since the whole testbed is placed on one machine with four cores, all virtual nodes compete for those four cores. A reasonable scheduling is only feasible within one virtual node but not among several. Processing responses related to the composition protocol, moreover, is a synchronized task, which consequently synchronizes all threads of the PCC node but not those of remote systems. With respect to the Five-Roles scenario, the overall testbed is able to more efficiently utilize the computing power of multiple threads simultaneously.

In the evaluation, no further complex structures have been emulated in this evaluation, though it was claimed that role-based models enable the composition of more complex application structures compared to, for instance, SOSSs. Thanks to the middleware architecture of RoleDiSCo, the challenges of complex application structures and full decentralization could be reduced. Since the composition process itself is centralized on demand, complex application structures can be composed similar as in a centralized system. In turn, RoleDiSCo enables every node in an infrastructure to become the coordinator for a particular pervasive collaboration. Complex composition structures, hence, solely depend on the *planner* used to determine the concrete composition plan. Developing a general-purpose planner was, however, not within the scope of this thesis as it is a research challenge of its own, which even holds true for application-specific planners that have to deal with a certain complexity of the application. Thus, the challenge of composition of complex application structures were broken down into the planning part and the composition part, the latter of which was addressed by the RoleDiSCo Middleware. As long as a planner provides the concrete instructions, the middleware will attempt to compose the given plan within the infrastructure.

As this thesis is solely concerned with composition and adaptation, any evaluation of the operational state was omitted as this can be restrained to remote procedure calls. Furthermore, the measurements regarding the operational phase would be of low

significance as the role-based runtime was mocked. According to Schütze and Castrillon [71], role-based runtimes behave very differently in terms of run-time performance. Static approaches, such as ObjectTeams [43] are faster than dynamic approaches such as LyRT [76] or SCROLL [60]. Hence, the run-time overhead during the operational phase strongly depends on the concrete role-based runtime.

In the experiments, solely isolated collaborations were considered, meaning that only one collaboration was composed and operational at one time. Though the investigation of how several collaborations would compose and operate in parallel is of interest, this is a barely controllable experiment. Even if multiple collaborations are instantiated simultaneously, composition will proceed in an uncontrolled way and presumably be performed sequentially. Thus, only the total composition time of all parallel collaborations could be measured, which would be a result of low significance.

The performance evaluation did not explicitly address adaptation. In this thesis, adaptation is considered a structural reconfiguration of an operational collaboration, which may occur whenever a new node joins the infrastructure or a collaborating node leaves the infrastructure. The planner calculates a new plan, new nodes are incorporated similar to them during initial composition and dropped nodes will receive a `Terminate` message and stop participating in the collaboration. Apart from minor differences, adaptation works the same way as composition. Thus, adaptation time in this context would be the recomposition time which comprises discovery, planning as well as actual recomposition. As these aspects were evaluated in isolation, it is in evidence that recomposition time is the result of those with respect to the number of changes, *i.e.*, added or removed roles and the number of nodes that are affected by those changes.

Lastly, the implementation of the RoleDiSCo Middleware offers possibilities for improvement as mentioned in Section 5.2 as, for instance, messaging in RoleDiSCo heavily relies on the serialization mechanisms provided by JGroups, the default one of Java, which is considered to be rather slow compared to custom implementations.

6.3 The ›*Role*‹ of Roles

This thesis aims for automated composition and adaptation of Smart Service Systems and thereby investigates whether role-based models are a suitable abstraction in order to achieve this. Especially the collaborative nature of the *Concept of Roles* was argued to be beneficial for automated composition of software systems. This section discusses the advantages and disadvantages of applying the *Concept of Roles* in this approach.

First, serendipity, *i.e.*, to integrate unforeseen entities or functionality at run time, is a key requirement of Smart Service Systems, challenging to be achieved with common component-based or service-oriented approaches as there building blocks are self-contained entities that cannot be split into the collaboration-dependent part and its concrete performance. Since players and roles are loosely coupled at both design and run time, their binding can be easily reconfigured thereby changing the overall behavior or integrating new behavior by providing a completely new player for an existing role instance. In the Distributed Slideshow scenario, it is conceivable that a new device is equipped with the Viewer role and joins an operating collaboration immediately. Thus, the *Concept of Roles* inherently allows for serendipity and for separation of concerns.

Second, Smart Service Systems establish collaborations among several, potentially similar, entities. The collaborative nature of roles desires a utilization for the composition of distributed systems as roles and their surrounding collaborations inherently specify relationships and thereby a complete system structure. In other words, the system is composed based on abstract functionalities, *i.e.*, the roles, whose actual performance is realized by entities that are not directly known to the composition, *i.e.*, the players. Compared to classic approaches, such as SOAs, roles are much more flexible. While in SOAs the service and its role essential to a business process, *i.e.*, the collaboration, are self-contained units, roles allow to split such a unit into the service, providing the base functionality and concrete performance, and its role essential to a collaboration.

Third, this separation allows for a demarcation of the development processes, which is a substantial achievement in order to integrate arbitrary systems considering the increasing amount of such. On the one hand, there are developers implementing services that provide a certain functionality but actually do not focus on how their services might integrate with others to create some synergy. On the other hand, there are developers who would like to integrate such services but are not able to capture all eventualities of concrete services that may occur. Applying the concept of roles, each of the groups can continue to solely focus on their own part. However, both or even a third party is able to integrate the individual parts by letting a service play a role in a certain collaboration.

Next, at run time, a role type can be instantiated several times for different collaborations, which allows for simultaneous participation of the player in different collaborations. In the experiments conducted afore, collaborations were considered to be isolated, *i.e.*, to be non-conflicting or non-competing. A real-world scenario, however, will for sure face this issue. Evidently, roles are beneficial in order to deal with isolation, as, just mentioned, a role type can be instantiated several times, played by the same player, thus having one physical resource participating in multiple collaborations. The crucial challenge

arises from physical resources that cannot be shared. Roles, however, are obviously able to abstract this issue to some extent and create some kind of resource scheduling, *i.e.*, roles on a single node rotate, managed locally by the role-based runtime, in order to share the physical resource. If this is not possible, such *competing collaborations* can only be addressed by *negotiating* a compromise among affected collaboration(s).

Conclusively, the *Concept of Roles* could act as enabling technology to integrate systems of different origins, types, architectural concepts, and granularity. This thesis showed that applying roles allows to easily compose a system out of potentially heterogeneous runtimes, such as a role-based runtime and a legacy one. Obviously, a runtime could be replaced with a SOA, a component-based system, a multi-agent system, or any other system that adopts a notion of roles. Multi-agent systems, for instance, employ roles to denote the current state or task of an agent. The role as a common denominator would enable agents of different vendors to collaborate, and moreover to integrate with other types of systems, such as SOAs or even local role-based runtimes.

A certain challenge is to rethink applications and systems the role-based way. This includes to become familiar with role-based modeling or programming languages and requires a broader range of tool support to develop role-based systems. For instance, the RoleDiSCo Development Support, cf. Section 5.1, does not result in a well-known role-based programming language. Thus, a holistic development support for role-based software systems is a remaining challenge. Besides this, developers are faced with the question which part of a functionality is abstract and which is concrete. The Distributed Slideshow scenario considered solely the collaborative parts, such as transmitting pictures in a serialized form, as abstract functionalities. In more complex applications, however, this will be a more challenging task. Colman and Han [19] addressed this challenge in a role-based fashion and defined several degrees of autonomy for roles and players. The collaboration specification adopted this idea partially but does not cover all the proposed degrees of freedom. This does not answer the initial question which part of the functionality is abstract and which is concrete, but at least enables the collaboration designers to offer some flexibility to the Phase ② developers.

6.4 Summary

This chapter was concerned with a quantitative and qualitative evaluation of the RoleDiSCo approach in order to show how the problems mentioned in Section 1.3 were addressed. The evaluation is a foundation for a review of the research questions

posed in Section 1.5, which will be comprehensively provided in Chapter 7. Hereinafter, the key findings of the evaluation are briefly summarized.

The case study used a sample application, *i.e.,* the Distributed Slideshow, which was specified using the *Role-based Collaboration Specification*. By design, the specification does not impose any restrictions on the players of the roles, which is a prerequisite in order to demarcate the development processes. Thereof, partial implementation was derived using the research prototype for the development support. Consequently, the utilization of the resulting artifact by the middleware, which is the second research prototype, was rigorously demonstrated. Lastly, achieving automated composition and adaptation in a decentralized environment was explained and implemented prototypically.

The RoleDiSCo Middleware's prototypical implementation's performance and scalability were evaluated. Performance was not a strong requirement, but time is generally a crucial metric. Hence, it is remarkable that the composition time stays below the mark of 1 s response time for interactive systems [62] even for system sizes up to 150 nodes although the implementation still offers possibilities for improvement. Moreover, the composition scales linearly to the number of nodes involved in the collaboration. So, automated composition was shown to be derivable from role-based models and composition itself was achieved in a considerably short amount of time. Consequently, the RoleDiSCo approach solved the problems of on-demand composition of smart service systems and of the discontinuity between design-time specification, and the run-time composition and adaptation.

Evidently, role-based models are a proper abstraction to specify complex service structures in order to realize on-demand composition and subsequent adaptation of Smart Service Systems at run time. The *RoleDiSCo Development Methodology* in conjunction with the *RoleDiSCo Middleware* allows to support the *Concept of Roles* throughout the entire system life cycle, thereby eliminating the discontinuity between design and run time. The development methodology, moreover, preserves the autonomy of both the autonomous service developer and the Smart Service System designer. Limitations of enforcing complex service structures in decentralized environments depend on the concrete planner that is used and the trade-off of applying role-based models in order to compose complex service structures in decentralized environments, in turn, is considerably small.

7 Conclusion

This chapter concludes the thesis and thereby briefly summarizes the preceding chapters. Thereafter, in conjunction with the main contributions of this thesis, the research questions and problem statements will be reviewed. Finally, remaining challenges and potential areas for future work are pointed out.

7.1 Summary

Chapter 1 motivated this thesis and argued that near-future smart computing environments, such as smart cities, smart homes, and in general the Internet of Things, will heavily rely on the spontaneous collaboration of independently developed services. However, existing solutions are limited in their application structure mainly due to an absence of overlying specifications or models. Moreover, such collaborations will be continuous or ongoing, and will not have a predefined expiration as, for instance, business processes have. The increasing amount of smart systems and their provided services requires new development approaches in order to demarcate the development processes of the individual services from that of the collaborations as much as possible. Eventually, approaches providing comprehensive models do not support spontaneous run-time composition and approaches achieving such do not utilize models supporting complex structures. The *Concept of Roles* was argued to provide an intuitive abstraction that perfectly matches the collaborative nature of Smart Service Systems. Hence, the approach was designed based on the hypothesis that *role-based abstractions allow to specify Smart Service Systems at design time and subsequently enable on-demand composition and adaptation of such systems in decentralized environments at run time*.

Chapter 2 briefly summarized the predominant role concepts available in literature, as no common definition of *roles* in computer science exists, and outlines the utilization and understanding of the *Concept of Roles* and its application within the scope of this thesis.

Chapter 3 comprehensively analyzes the current state of practice and related work. Approaches in the domain of role-based modeling, role-based runtimes and systems achieving spontaneous collaborations, notably Self-Organizing Software Systems, were analyzed. Besides seeing the problem statements confirmed, the discontinuity between design and run time was in evidence. The investigated approaches either utilize comprehensive models, partially even role-based models, but do not achieve automated composition, or they achieve automated composition of rather simple, predefined structures.

Subsequently, Chapter 4 presents the concepts to solve the aforementioned problems. First, the *RoleDiSCo Development Methodology* separates the development of the overall Smart Service System, *i.e.,* the collaboration, from that of the service and its role essential to the collaboration. The methodology includes a *Role-based Collaboration Specification* to specify Smart Service Systems. Next, the *RoleDiSCo Middleware Architecture* is designed to utilize the artifacts resulting from the development methodology in order to achieve on-demand composition of Smart Service Systems in decentralized environments. The middleware automatically derives discovery information, provides a respective decentralized discovery mechanism and abstracts concrete underlying network infrastructures. Finally, in order to compose and adapt such systems coordinately, a protocol for *coordinated on-demand composition and subsequent adaptation* is proposed. Chapter 5 provides some insights into the implementational details of the two research prototypes that were developed in order to evaluate the approach.

Lastly, Chapter 6 is concerned with the qualitative and quantitative evaluation of the RoleDiSCo approach. A case study rigorously demonstrates the complete development methodology, how independent services can be integrated in both a role-based and a non-role-based way, and eventually how these artifacts are used by the middleware in order to achieve automated discovery, composition and adaptation in a coordinated manner. Additionally, the approach was evaluated with respect to performance and scalability. The evaluation concludes with a critical discussion concerning the utilization of the *Concept of Roles* for on-demand composition of Smart Service Systems.

7.2 Research Results

In Section 1.3, four key problems of engineering Smart Service Systems were identified, all with the challenges of heterogeneity, context-awareness, and decentralization in mind: on-demand composition of Smart Service Systems; a missing context-aware, complex service system specification; a discontinuity from design-time specification to

run-time composition and adaptation; and the intertwined development processes of autonomous services and their role essential to a collaboration. The remainder of this section is concerned with answering the research questions posed in Section 1.5.

1. *What is a proper abstraction to specify complex service collaborations in order to realize on-demand composition and subsequent adaptation of Smart Service Systems at run time?*

The *Concept of Roles* is a proper abstraction to specify complex service collaborations and to realize on-demand composition as well as adaptation thereof. Roles inherently allow for serendipity, *i.e.,* to integrate unforeseen entities or functionality at run time, which is a huge advantage in order to realize heterogeneity. Since players and roles are loosely coupled, their binding can be easily reconfigured thereby changing the overall behavior or integrating new behavior by providing a new player for an existing role.

The collaborative nature of roles desires a utilization for the composition of distributed systems as roles and their surrounding collaborations inherently specify relationships and thereby a complete system structure. Compared to classic approaches, *e.g.,* SOAs, roles are much more flexible. While, for instance, in SOAs the service and its role essential to a complete business process result in the same logical and physical unit, applying the concept of roles allows to split this unit into the service, providing the base functionality and concrete execution, and its role essential to a collaboration.

This separation allows to demarcate the development processes of the individual services from those of the collaborations. Developers implementing services that provide a certain functionality do not have to focus on how their services might integrate with others. Conversely, developers who would like to integrate such services do not have to capture all eventualities of concrete services to be integrated. Applying the concept of roles, each of the groups can continue to focus on its own part. Both or even a third party is able to integrate the individual parts by letting a service play a role in a collaboration, which is a substantial achievement considering the increasing amount of smart systems.

At run time, a role type can be instantiated several times for different collaborations, which allows for simultaneous participation of the player in different collaborations. Thus, roles are very beneficial in order to deal with isolation as multiple role instances can be played by the same player, thereby having one physical resource participating in multiple collaborations. Roles, moreover, allow to abstract physical resources that cannot be shared to some extent: multiple role instances on a single node may rotate in order to share the physical resource, thereby creating some kind of resource scheduling.

Conclusively, the *Concept of Roles* could act as an enabling technology to integrate systems of different origins, types, architectural concepts, and granularity. This thesis showed that applying roles allows to easily compose a system out of potentially heterogeneous runtimes, such as a role-based runtime and a legacy one. Obviously, a runtime could be replaced with a SOA, a component-based system, a multi-agent system, or any other kind of system that adopts a notion of roles. Nevertheless, the role as a common denominator could enable systems of different vendors to collaborate, and moreover to integrate with other systems of other types, such as SOAs or even a local role-based runtime.

2. *How can these abstractions be supported throughout the application life cycle (i.e., development and operation phase) in a way which preserves the autonomy of both the autonomous service designer and the Smart Service System designer?*

In order to support the *Concept of Roles* throughout the application life cycle, the *RoleDiSCo Development Methodology*, which demarcates the development processes and thereby preserves the individual developers' independence, results in artifacts that are designed to be used by the *RoleDiSCo Middleware*, which thereby eliminates the existing discontinuity between design and run time, and complements the support of the *Concept of Roles* throughout the entire application life cycle.

The *RoleDiSCo Development Methodology*, a two-phase development methodology, demarcates the development of the service from that of its role essential to a collaboration: First, a collaboration designer specifies the overall collaboration including its abstract functionality using the proposed role-based, context-aware collaboration specification. This specification is independent of the roles' potential players. Thereof, a partial implementation is derived, which is later complemented with its concrete performance by several other developers in a second phase. The derived partial implementation can be integrated without changing the existing implementation. This preserves the individual development lanes of the autonomous services and those of the collaborations. The artifacts resulting from the development methodology are designed to be used by the *RoleDiSCo Middleware*, thereby eliminating the discontinuity between design and run time, and supporting the *Concept of Roles* throughout the entire application life cycle.

3. *What are limitations of enforcing complex service structures in decentralized environments?*

Thanks to the RoleDiSCo Middleware, the composition and adaptation process is only limited by the respective planner that calculates the composition plan. The prototypical implementation lacks a general-purpose planner, which would be able to compute a composition plan based on an arbitrary collaboration specification. In other words, limitations concerning the service structure only arise from the designated planner

but neither from the composition protocol nor from the proposed middleware. At run time, the system will probably not scale indefinitely, which may be caused by various reasons, such as limited memory on the Pervasive Collaboration Coordinator, or limited network bandwidth. Additionally, large-scale collaborations might face volatile network situations causing a lot of adaptations. Though the system will continue to work, the overall stability and usability might be affected. Nevertheless, the evaluation demonstrated that the approach scales reasonably well up to 150 nodes, which was an upper boundary solely due to limited resources of the machine used for the experiments.

4. *What is the trade-off of applying the concept of roles to on-demand composition and adaptation of Smart Service Systems? What are limitations?*

Roles primarily exist as types while services or components are already instantiated and run. Thus, roles have to be instantiated first, then they have to be bound to their players, and eventually that binding is to be activated, which is considered equivalent to *binding* a service to another. The performance evaluation, however, shows that the protocol's overhead in total is significantly low. Of course, if role instantiation, player instantiation, or their binding is more time-consuming, the overall performance will decrease. Since the total composition time for all three steps in two scenarios with 150 nodes, thus 150 respectively 750 role instances, is less than half a second, the trade-off of applying the role concept to on-demand composition and adaptation is considerably small. The case study argued the role-based approach to be feasible and beneficial, while the performance evaluation demonstrated the composition protocol's overhead introduced by roles to be of little significance. In other words, automated composition and structural adaptation can be derived from role-based models almost for free.

7.3 Future Work

This thesis combined two different research domains. One goal was to utilize the comprehensive models from the domain of Role-based Software Systems, with autonomous composition and adaptation features available in Self-Organizing Software Systems. Notwithstanding its contribution to the current state of research, a couple of interesting research challenges were not addressed within this thesis.

General-Purpose Planner

A crucial limitation in order to realize arbitrary intricate application structures is the underlying planner, which matches a collaboration specification with the respective

discovery information in order to provide the composition instructions. Section 5.2.3 pointed out that such planners exist [72] and explained how to utilize them. As a logical consequence, it has to be investigated, whether such a planner should compute plans based on a yet to be defined composition plan metamodel, a model derived from the specification as part of the proposed development methodology, or on a manually provided model, which equates to manually implementing the planner. Assuming that it is possible to calculate plans either on a static metamodel (general-purpose planner) or a derived model (generated, tailored planner), it would be interesting to investigate whether a general-purpose planner is significantly slower than a tailored one.

Formally Founded Specification

Section 3.1.2 discussed the Compartment-Role-Object Model (CROM) [56] as a formal metamodel capturing many features of role-based modeling languages existent in literature. Though the specification is inspired by CROM, it is not formally founded on that. CROM is based on ontological foundations of roles, which were not discussed in this thesis, but have several implications on the definitions of players, roles, and compartments (*i.e.*, collaborations). Additionally, the player's type is always known and predefined in CROM, which was to be avoided in favor of serendipity. Assuming the player's interface, generated for the player as part of the code generation process, as the player's type, the ontological foundations are still violated as an interface is not *rigid*, but a CROM-based formal foundation appears achievable. This would ease to interlink approaches employing different notions of roles.

Fine-Grained Adaptation

As of now, adaptation is considered as a structural recomposition caused by infrastructural or contextual changes. Evidently, adaptation should be much more fine-grained. Since roles are stateful, decoupled units, they can easily be migrated among players. However, currently the approach is not able to automatically derive a migration from change sets that result from updated composition plans, but limited to adding and removing roles from the collaboration. This issue has been partially explored in [MW6] by integrating an approach that reliably performs complex run-time adaptations of highly distributed software systems without a central control unit [82] into the approaches presented in this thesis. Thereby, the proposed specification was extended by particular instructions that enable the middleware to migrate roles specifically. Since such adaptations usually depend on a concrete scenario, it is reasonable that the collaboration designer will take care of such explicit adaptations.

The End.

Bibliography

[1] Wil M P van der Aalst and Arthur H M ter Hofstede. "YAWL: Yet Another Workflow Language". In: *Information Systems* 30.4 (2005), pp. 245–275. DOI: 10.1016/j.is.2004.02.002.

[2] *Android Developers*. Android. URL: https://developer.android.com (visited on 10/07/2017).

[3] *Avahi Service Discovery Suite*. URL: http://avahi.org (visited on 02/24/2017).

[4] Charles W Bachman and Manilal Daya. "The Role Concept in Data Models". In: *Proceedings of the Third International Conference on Very Large Data Bases*. VLDB Endowment, Oct. 1977.

[5] Bela Ban. *JGroups. A Toolkit for Reliable Messaging*. Red Hat. 2017. URL: http://www.jgroups.org/ (visited on 09/10/2017).

[6] Sergio Barile and Francesco Polese. "Smart Service Systems and Viable Service Systems: Applying Systems Theory to Service Science". In: *Service Science* 2.1-2 (2010), pp. 21–40. DOI: 10.1287/serv.2.1_2.21.

[7] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Learn How to Implement a DSL with Xtext and Xtend Using Easy-to-Understand Examples and Best Practices. Packt Publishing Ltd, Aug. 2016. ISBN: 9781786464965.

[8] Lachlan Birdsey, Claudia Szabo, and Katrina Falkner. "CASL: A declarative domain specific language for modeling Complex Adaptive Systems". In: *2016 Winter Simulation Conference (WSC)*. IEEE, 2016, pp. 1241–1252. ISBN: 978-1-5090-4486-3. DOI: 10.1109/WSC.2016.7822180.

[9] Guido Boella and Friedrich Steimann. "Roles and Relationships". In: *ECOOP'07: Proceedings of the 2007 Conference on Object-oriented Technology*. Berlin, Heidelberg: Springer, July 2007.

[10] *Bonjour*. Apple, Inc. URL: https://support.apple.com/de-de/bonjour (visited on 02/24/2017).

[11] Antonio Bucchiarone, Antonio Cicchetti, and Martina De Sanctis. "Towards a Domain Specific Language for Engineering Collective Adaptive Systems". In: *2017 IEEE International Workshops on Foundations and Applications of Self* Systems* (Tucson, AZ, USA, Sept. 18–22, 2017). Sept. 2017. DOI: 10.1109/FAS-W.2017.115.

[13] Tomas Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. "DEECO: An Ensemble-based Component System". In: *Proceedings of the 16th International ACM Sigsoft Symposium on Component-based Software Engineering*. New York, NY, USA: ACM, 2013, pp. 81–90. ISBN: 978-1-4503-2122-8. DOI: 10.1145/2465449.2465462.

[12] Tomas Bures, Filip Krijt, Frantisek Plasil, Petr Hnetynka, and Zbynek Jiracek. "Towards Intelligent Ensembles". In: *Proceedings of the 2015 European Conference on Software Architecture Workshops* (Dubrovnik, Cavtat, Croatia, Sept. 7–11, 2015). ECSAW '15. New York, NY, USA: ACM, 2015, 17:1–17:4. ISBN: 978-1-4503-3393-1. DOI: 10.1145/2797433.2797450.

[14] *Business Process Model and Notation*. OMG Specification. Version 2.0. Object Management Group (OMG), Jan. 3, 2011. URL: http://www.omg.org/spec/BPMN/2.0/ (visited on 02/10/2017).

[15] Mauro Caporuscio and Carlo Ghezzi. "Engineering Future Internet Applications: The Prime Approach". In: *Journal of Systems and Software* 106.August 2015 (Apr. 2015), pp. 9–27. DOI: 10.1016/j.jss.2015.03.102.

[16] Mauro Caporuscio, Vincenzo Grassi, Moreno Marzolla, and Raffaela Mirandola. "GoPrime: A Fully Decentralized Middleware for Utility-Aware Service Assembly". In: *Software Engineering, IEEE Transactions on* 42.2 (Feb. 2016), pp. 136–152. DOI: 10.1109/TSE.2015.2476797.

[17] Valeria Cardellini, Mirko D'Angelo, Vincenzo Grassi, Moreno Marzolla, and Raffaela Mirandola. "A Decentralized Approach to Network-Aware Service Composition". In: *Service Oriented and Cloud Computing*. Cham: Springer, Cham, Sept. 2015, pp. 34–48. ISBN: 978-3-319-24071-8. DOI: 10.1007/978-3-319-24072-5_3.

[18] Alan W Colman. "Role Oriented Adaptive Design". PhD thesis. Swinburne University of Technology, 2006.

[19] Alan W Colman and Jun Han. "On the Autonomy of Software Entities and Modes of Organisation". In: *Proceedings of the 1st International Workshop on Coordination and Organisation*. 2005. URL: http://www.ict.swin.edu.au/personal/jhan/jhanPapers/coorg05autonomy.pdf.

[20] Alan W Colman and Jun Han. "Roles, Players and Adaptable Organizations". In: *Applied Ontology* 2.2 (Apr. 2007), pp. 105–126.

[21] Alan W Colman and Jun Han. "Using Role-Based Coordination to Achieve Software Adaptability". In: *Science of Computer Programming* 64.2 (Jan. 2007), pp. 223–245. DOI: 10.1016/j.scico.2006.06.006.

[22] Anind K Dey. "Understanding and Using Context". In: *Personal Ubiquitous Comput.* 5.1 (Jan. 2001), pp. 4–7. DOI: 10.1007/s007790170019.

[23] Giovanna Di Marzo Serugendo and J Fitzgerald. "MetaSelf: an Architecture and a Development Method for Dependable Self-* Systems". In: *SAC'10: Proceedings of the 2010 ACM Symposium on Applied Computing*. 2010.

[24] Giovanna Di Marzo Serugendo et al. "Self-Organisation: Paradigms and Applications". In: *Engineering Self-Organising Systems*. Ed. by Giovanna Di Marzo Serugendo, Anthony Karageorgos, Omer F Rana, and Franco Zambonelli. Berlin, Heidelberg: Springer, 2004, pp. 1–19.

[25] *Eclipse Modeling Framework (EMF)*. The Eclipse Foundation. URL: http://www.eclipse.org/modeling/emf/ (visited on 08/11/2017).

[26] Sven Efftinge et al. "Xbase: Implementing Domain-specific Languages for Java". In: *Proceedings of the 11th International Conference on Generative Programming and Component Engineering* (Dresden, Germany, Sept. 26–27, 2012). GPCE '12. New York, NY, USA: ACM, 2012, pp. 112–121. ISBN: 978-1-4503-1129-8. DOI: 10.1145/2371401.2371419.

[27] eMarketer Inc. *2 Billion Consumers Worldwide to Get Smart(phones) by 2016*. Dec. 2014. URL: http://www.emarketer.com/Article/2-Billion-Consumers-Worldwide-Smartphones-by-2016/1011694 (visited on 01/05/2016).

[28] Roy Thomas Fielding and Richard N Taylor. "Principled design of the modern Web architecture". In: *ACM Transactions on Internet Technology (TOIT)* 2.2 (May 2002), pp. 115–150. DOI: 10.1145/514183.514185.

[29] Martin Fowler. *Inversion of Control Containers and the Dependency Injection Pattern*. Jan. 23, 2004. URL: https://martinfowler.com/articles/injection.html (visited on 10/05/2017).

[30] Pedro Garcia Lopez et al. "Edge-centric Computing: Vision and Challenges". In: *SIGCOMM Comput. Commun. Rev.* 45.5 (Sept. 2015), pp. 37–42. ISSN: 0146-4833. DOI: 10.1145/2831347.2831354.

[31] Gartner, Inc. *Gartner Says 6.4 Billion Connected "Things" Will Be in Use in 2016, Up 30 Percent From 2015*. Press Release. Nov. 2015. URL: http://www.gartner.com/newsroom/id/3165317.

[32] Gartner, Inc. *Gartner Says a Typical Family Home Could Contain More Than 500 Smart Devices by 2022*. Press Release. Sept. 2014. URL: http://www.gartner.com/newsroom/id/2839717.

[33] Gartner, Inc. *Gartner's 2016 Hype Cycle for Emerging Technologies Identifies Three Key Trends That Organizations Must Track to Gain Competitive Advantage*. Aug. 2016. URL: http://www.gartner.com/newsroom/id/3412017.

[34] Valerio Genovese. "A Meta-model for Roles: Introducing Sessions". In: *Proceedings of the 2nd Workshop on Roles and Relationships in Object Oriented Programming, Multiagent Systems, and Ontologies*. Ed. by Guido Boella, Steffen Goebel, Friedrich Steimann, Steffen Zschaler, and Michael Cebulla. July 2007, pp. 27–38. URL: http://www.di.unito.it/~genovese/publications/2007/roles07.pdf.

[35] Yaron Y. Goland, Ting Cai, Paul Leach, Ye Gu, and Shivaun Albright. *Simple Service Discovery Protocol/1.0*. Tech. rep. Oct. 28, 1999. URL: ftp://ftp.pwg.org/pub/pwg/ipp/new_SSDP/draft-cai-ssdp-v1-03.txt (visited on 01/05/2017).

[36] *Google Chromecast*. Google Inc. 2016. URL: https://www.google.com/intl/en_us/chromecast/ (visited on 01/05/2017).

[37] *Google Guice*. Version 4.1. Google Inc. 2017. URL: https://github.com/google/guice (visited on 10/05/2017).

[38] Joel Greenyer, Larissa Chazette, Daniel Gritzner, and Eric Wete. "A Scenario-Based MDE Process for Dynamic Topology Collaborative Reactive Systems – Early Virtual Prototyping of Car-to-X System Specifications". In: *Proceedings Workshops zur Modellierung in der Entwicklung von kollaborativen eingebetteten Systemen (MEKES) (to appear)*. 2018. URL: http://jgreen.de/wp-content/documents/2018/virtual-prototyping-of-car-to-x-applications-camera-ready.pdf.

[39] *Guava. Google Core Libraries for Java*. Version 22.0. Google Inc. 2017. URL: https://github.com/google/guava (visited on 10/05/2017).

[40] Erik Guttman. "Autoconfiguration for IP networking: enabling local communication". In: *IEEE Internet Computing* 5.3 (2001), pp. 81–86. DOI: 10.1109/4236.935181.

[41] Robrecht Haesevoets, Danny Weyns, and Tom Holvoet. "Architecture-centric Support for Adaptive Service Collaborations". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23.1 (Feb. 2014), pp. 2–40.

[42] Rolf Hennicker and Annabelle Klarl. "Foundations for Ensemble Modeling – The Helena Approach". In: *Specification, Algebra, and Software*. Berlin, Heidelberg: Springer, 2014, pp. 359–381. ISBN: 978-3-642-54623-5. DOI: 10.1007/978-3-642-54624-2_18.

[43] Stephan Herrmann. "Object Teams: Improving Modularity for Crosscutting Collaborations". In: *Objects, Components, Architectures, Services, and Applications for a Networked World: International Conference NetObjectDays, NODe 2002 Erfurt, Germany, October 7–10, 2002 Revised Papers*. Ed. by Mehmet Aksit, Mira Mezini, and Rainer Unland. Berlin, Heidelberg: Springer, 2003, pp. 248–264.

[44] Stephan Herrmann, Christine Hundt, Katharina Mehner, and Jan Wloka. "Using Guard Predicates for Generalized Control of Aspect Instantiation and Activation". In: *DAW'05: Dynamic Aspects Workshop*. 2005, pp. 93–101. URL: http://www.objectteams.org/publications/DAW05.pdf.

[45] *How to AirPlay content from your iPhone, iPad, or iPod touch*. Apple Inc. Sept. 13, 2016. URL: https://support.apple.com/en-us/HT204289 (visited on 01/05/2017).

[46] Tobias Jäkel, Martin Weißbach, Kai Herrmann, Hannes Voigt, and Max Leuthäuser. "Position Paper: Runtime Model for Role-Based Software Systems". In: *2016 IEEE International Conference on Autonomic Computing (ICAC)*. July 2016, pp. 380–387. DOI: 10.1109/ICAC.2016.17.

[47] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. "Gossip-based Aggregation in Large Dynamic Networks". In: *ACM Transactions on Computer Systems (TOCS)* 23.3 (Aug. 2005), pp. 219–252. DOI: 10.1145/1082469.1082470.

[48] David Kempe, Alin Dobra, and Johannes Gehrke. "Gossip-Based Computation of Aggregate Information". In: *44th Annual IEEE Symposium on Foundations of Computer Science - FOCS 2003*. IEEE Computer. Soc, 2003, pp. 482–491. ISBN: 0-7695-2040-5. DOI: 10.1109/SFCS.2003.1238221.

[49] Jaroslav Keznikl, Tomas Bures, Frantisek Plasil, and Michal Kit. "Towards Dependable Emergent Ensembles of Components: The DEECo Component Model". In: *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. IEEE, pp. 249–252. ISBN: 978-0-7695-4827-2. DOI: 10.1109/WICSA-ECSA.212.39.

[50] Annabelle Klarl. "Engineering Self-Adaptive Systems with the Role-Based Architecture of Helena". In: *24th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE, Aug. 2015, pp. 3–8. ISBN: 978-1-4673-7692-1. DOI: 10.1109/WETICE.2015.32.

[51] Annabelle Klarl, Lucia Cichella, and Rolf Hennicker. "From Helena Ensemble Specifications to Executable Code". In: *Formal Aspects of Component Software*. Cham: Springer Intl. Publ., Sept. 2014, pp. 183–190. ISBN: 978-3-319-15316-2. DOI: 10.1007/978-3-319-15317-9_11.

[52] Annabelle Klarl and Rolf Hennicker. "Design and Implementation of Dynamically Evolving Ensembles with the Helena Framework". In: *2014 23rd Australian Software Engineering Conference (ASWEC)*. IEEE, 2014, pp. 15–24. ISBN: 978-1-4799-3149-1. DOI: 10.1109/ASWEC.2014.26.

[53] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. "A Survey on Engineering Approaches for Self-Adaptive Systems". In: *Pervasive and Mobile Computing* 17 (Feb. 2015), pp. 184–206. DOI: 10.1016/j.pmcj.2014.09.009.

[54] Thomas Kühn. *formalCROM – Implementation*. URL: https://github.com/Eden-06/formalCROM (visited on 03/24/2017).

[55] Thomas Kühn, Kay Bierzynski, Sebastian Richly, and Uwe Aßmann. "FRaMED: Full-fledge Role Modeling Editor (Tool Demo)". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. New York, NY, USA: ACM, Oct. 2016, pp. 132–136. ISBN: 978-1-4503-4447-0. DOI: 10.1145/2997364.2997371.

[56] Thomas Kühn, Stephan Böhme, Sebastian Götz, and Uwe Aßmann. "A Combined Formal Model for Relational Context-dependent Roles". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering* (Pittsburgh, PA, USA). SLE 2015. New York, NY, USA: ACM, 2015, pp. 113–124. ISBN: 978-1-4503-3686-4. DOI: 10.1145/2814251.2814255.

[57] Thomas Kühn, Stephan Böhme, Sebastian Götz, and Uwe Aßmann. *A Combined Formal Model for Relational Context-Dependent Roles (Extended)*. Tech. rep. TUD-FI15-04-September 2015. Dresden: Technische Universität Dresden, Sept. 2015. URL: http://www.qucosa.de/urnnbn/urn:nbn:de:bsz:14-qucosa-178506.

[58] Thomas Kühn, Max Leuthäuser, Sebastian Götz, Christoph Seidl, and Uwe Aßmann. "A Metamodel Family for Role-Based Modeling and Programming Languages". In: *Software Language Engineering*. Cham: Springer Intl. Publ., Sept. 2014, pp. 141–160. ISBN: 978-3-319-11244-2. DOI: 10.1007/978-3-319-11245-9_8.

[59] P. Leach, M. Mealling, and R. Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. IETF, July 2005, pp. 1–32. URL: https://tools.ietf.org/html/rfc4122.

[60] Max Leuthäuser and Uwe Aßmann. "Enabling View-based Programming with SCROLL". In: *Proceedings of the 2015 Joint MORSE/VAO Workshop*. New York, NY, USA: ACM Press, 2015, pp. 25–33.

[61] Mengchi Liu and Jie Hu. "Information Networking Model". In: *Proceedings of the 28th International Conference on Conceptual Modeling*. Conceptual Modeling - ER 2009 (Gramado, Brazil, Nov. 9–12, 2009). Ed. by Alberto H. F. Laender, Silvana Castano, Umeshwar Dayal, Fabio Casati, and José Palazzo M. de Oliveira. Berlin, Heidelberg: Springer, 2009. ISBN: 978-3-642-04839-5. DOI: 10.1007/978-3-642-04840-1_12.

[62] Jakob Nielsen. *Usability Engineering*. San Francisco, CA: Morgan Kaufmann, 1993. ISBN: 978-0-12-518406-9.

[63] Open Connectivity Foundation, Inc. *UPnP Device Architecture 2.0*. Feb. 2015. URL: http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v2.0.pdf.

[64] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd ed. Dallas, Texas ; Raleigh, North Carolina : The Pragmatic Bookshelf, Sept. 2014. ISBN: 1934356999.

[65] Christian Piechnick. “Smart Application Grids”. Diploma Thesis. Dresden, Oct. 2011.

[66] Christian Piechnick, Sebastian Richly, Sebastian Götz, Claas Wilke, and Uwe Aßmann. “Using Role-Based Composition to Support Unanticipated, Dynamic Adaptation – Smart Application Grids”. In: *ADAPTIVE 2012: The Fourth International Conference on Adaptive and Self-Adaptive Systems and Applications*. 2012.

[67] Sławek Piotrowski. *ClassIndex*. Version 3.4. Atteo. 2017. URL: https://github.com/atteo/classindex (visited on 10/05/2017).

[68] Trygve Reenskaug and Jim Coplien. *The DCI Architecture: A New Vision of Object-Oriented Programming*. May 2009. URL: http://www.artima.com/articles/dci_vision.html.

[69] Dirk Riehle and Thomas Gross. “Role Model Based Framework Design and Integration”. In: *ACM SIGPLAN Notices* (1998). DOI: 10.1145/286936.286951.

[70] Mazeiar Salehie and Ladan Tahvildari. “Self-adaptive Software: Landscape and Research Challenges”. In: *ACM Trans. Auton. Adapt. Syst.* 4.2 (May 2009), 14:1–14:42. ISSN: 1556-4665. DOI: 10.1145/1516533.1516538.

[71] Lars Schütze and Jeronimo Castrillon. “Analyzing State-of-the-Art Role-based Programming Languages”. In: *Companion to the First International Conference on the Art, Science and Engineering of Programming* (Brussels, Belgium). Programming ’17. New York, NY, USA: ACM, 2017, 9:1–9:6. ISBN: 978-1-4503-4836-2. DOI: 10.1145/3079368.3079386.

[72] Geoffrey De Smet et al. *OptaPlanner User Guide*. Optaplanner: An Open Source Constraint Satisfaction Solver in Java. Red Hat et al. URL: https://www.optaplanner.org.

[73] Friedrich Steimann. “On the Representation of Roles in Object-oriented and Conceptual Modelling”. In: *Data & Knowledge Engineering* 35.1 (Oct. 2000), pp. 83–106. DOI: 10.1016/S0169-023X(00)00023-9.

[74] Friedrich Steimann. “Role = Interface: A Merger of Concepts”. In: (2001). ISSN: 0896-8438. URL: http://deposit.fernuni-hagen.de/2183/.

[75] Daniel Sykes, Jeff Magee, and Jeff Kramer. “FlashMob: Distributed Adaptive Self-assembly”. In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. New York, NY, USA: ACM, May 2011, pp. 100–109.

[76] Nguonly Taing, Thomas Springer, Nicolás Cardozo, and Alexander Schill. “A Dynamic Instance Binding Mechanism Supporting Run-time Variability of Role-based Software Systems”. In: *Companion Proceedings of the 15th International Conference on Modularity* (Málaga, Spain). MODULARITY Companion 2016. New York, NY, USA: ACM, 2016, pp. 137–142. ISBN: 978-1-4503-4033-5. DOI: 10.1145/2892664.2892687.

[77] Luis M Vaquero and Luis Rodero-Merino. “Finding your Way in the Fog”. In: *ACM SIGCOMM Computer Communication Review* 44.5 (Oct. 2014), pp. 27–32. DOI: 10.1145/2677046.2677052.

[78] *Web Services Business Process Execution Language*. OASIS Standard. Version 2.0. OASIS, Apr. 11, 2007. URL: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (visited on 02/10/2017).

[79] *Web Services Choreography Description Language*. W3C Candidate Recommendation. Version 1.0. World Wide Web Consortium (W3C), Nov. 9, 2005. URL: http://www.w3.org/TR/ws-cdl-10/ (visited on 02/10/2017).

[80] *Web Services Description Language*. W3C Recommendation. Version 2.0. World Wide Web Consortium (W3C), June 26, 2007. URL: http://www.w3.org/TR/wsdl20 (visited on 02/10/2017).

[81] Mark Weiser. “The Computer for the 21st Century”. In: *Scientific American* 265.3 (1991), pp. 94–104.

[82] Martin Weißbach and Thomas Springer. “Coordinated Execution of Adaptation Operations in Distributed Role-based Software Systems”. In: *SAC’17* (Marrakesh, Marocco). SAC ’17. New York, NY, USA: ACM, 2017, pp. 45–50. DOI: 10.1145/3019612.3019624.

[83] Danny Weyns, Sam Malek, and Jesper Andersson. "On Decentralized Self-Adaptation: Lessons From the Trenches and Challenges for the Future". In: *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. New York, NY, USA: ACM, May 2010, pp. 84–93. ISBN: 978-1-60558-971-8. DOI: 10.1145/1808984.1808994.

[84] Danny Weyns et al. "On Patterns for Decentralized Control in Self-Adaptive Systems". In: *Software Engineering for Self-Adaptive Systems II*. Ed. by Rogério de Lemos, Holger Giese, Hausi A Müller, and Mary Shaw. Berlin, Heidelberg: Springer, 2013, pp. 76–107.

[85] Gerd Wütherich, Nils Hartmann, Bernd Kolb, and Matthias Lübken. *Die OSGi Service Platform*. Eine Einführung mit Eclipse Equinox. Heidelberg: dpunkt.verlag, 2008. ISBN: 9783898644570.

[86] *Xtext – Language Engineering Made Easy!* The Eclipse Foundation. URL: http://www.eclipse.org/Xtext/ (visited on 10/08/2017).

List of Figures

List of Tables

List of Listings

Appendix A Supplementary Figures & Tables

Discovery Time Experiments

The data obtained during the experiments regarding the discovery time (Section 6.2.2) for the Five-Roles scenario was adjusted by considering all values larger than 2 s as outliers. Figure A.1 displays the average discovery time based on the original, unadjusted data. Evidently, starting after 55 nodes, the discovery time constantly increases until a threshold of roughly 2.75 s is reached. In contrast to that, the dashed line represents the average discovery time based on the adjusted data, similar to Figure 6.7.

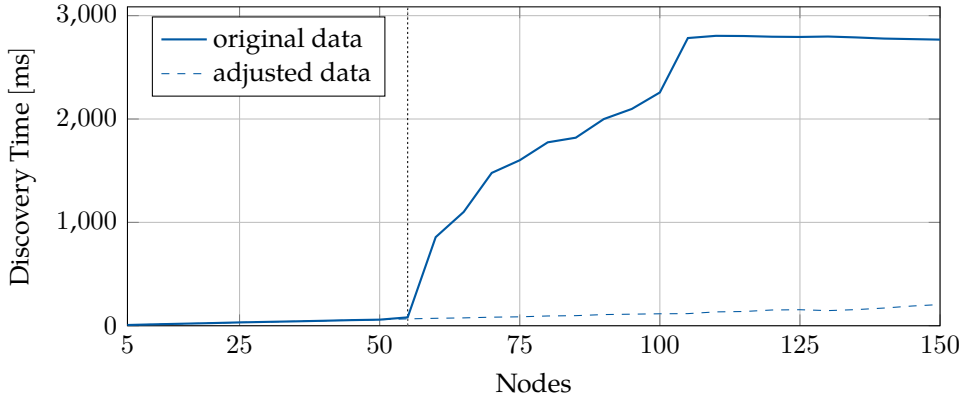


Figure A.1: Comparison of the average Discovery Time based on Original and Adjusted Data of the Five-Roles Scenario's Experiments.

In Figure A.2, the original data set is grouped into intervals similar to a histogram. Up to 50 nodes, technically all of the 1000 values measured are below the mark of 1 s, meaning that the discovery process was completed within 1 s. Above 50 nodes, however, the values lie within the range of 2 s to 3 s. only a very few results lie within the intervals of 1 s to 2 s, 3 s to 4 s, and 4 s to 5 s. The reason for this gap is related to the JGroups' issue discussed in Section 5.2.1, where messages got lost and had to be resent, which introduced a certain delay and thereby caused this gap. The reason that messages have to be resent in this particular case is that the physical resources of the testbed are limited and network packets are dropped physically. The first increase

is noticeable at 55 nodes and indicates that an upper boundary for reliably sending and receiving messages is reached. At this point, 324 messages had to be processed by the node under test. The experiments concerning the discovery time of the Simple-Sample scenario and the composition time in general are not affected by this issue as the total number of messages sent or received rather simultaneously never exceeds a total of 150 messages for the composition and a total of 300 messages for the discovery process of the Simple-Sample scenario. Consequently, all values larger than 2 s were considered outliers as they are caused by the limited resources of the testbed and not by the approach itself. Though this weakens the significance of the results, especially of those for larger sets of nodes, the adjusted data provides a much better insight into the behavior of the discovery process than the original data set. In order not to distort the overall results, those values were ignored when calculating the average discovery time for both scenarios in Figure 6.7.

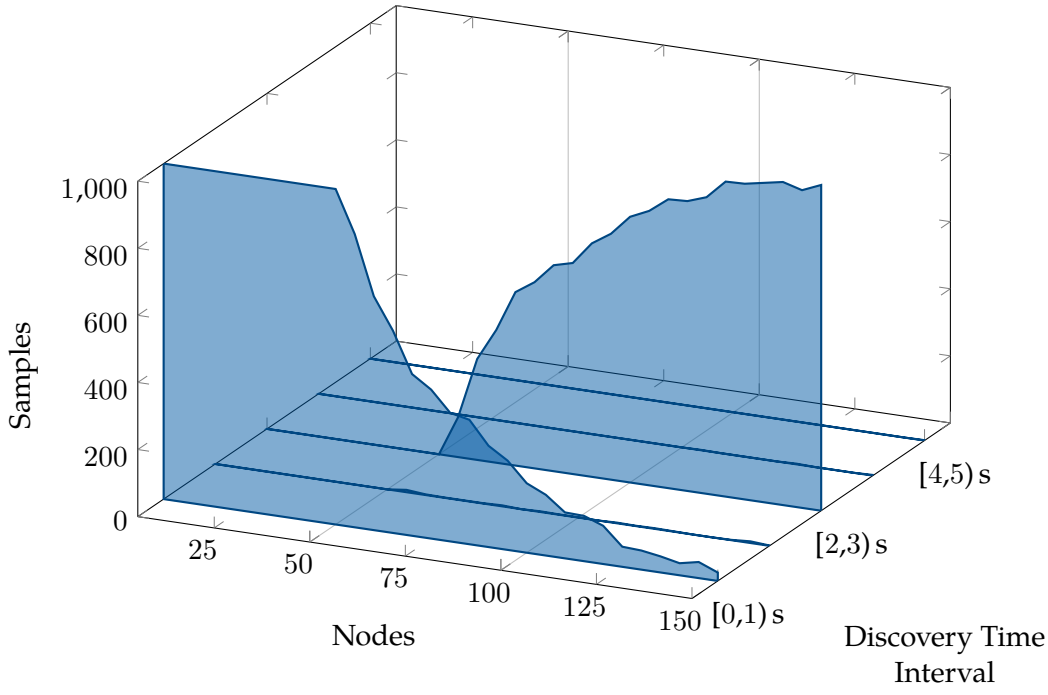


Figure A.2: Discovery Time Data Points Accumulated by Intervals.

Composition Time Experiments

Figures A.3 and A.4 address the issue that the Five-Roles scenario allegedly performs better than the Simple-Sample in terms of the composition time. Evidently, the Simple-

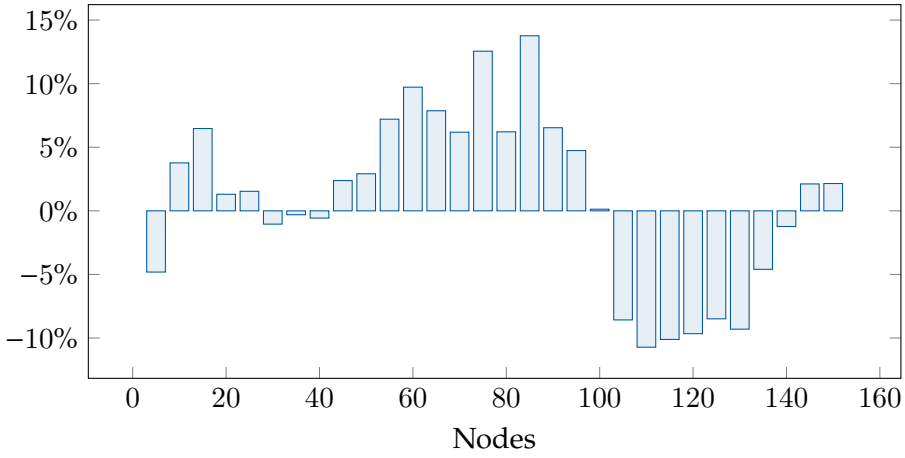


Figure A.3: Composition Time: Five-Roles Scenario in relation to the Simple-Sample (baseline).

Sample scenario performs worse compared to the Five-Roles scenario, especially for system sizes from 105 to 140 nodes. In Figure A.4, the composition time is segmented

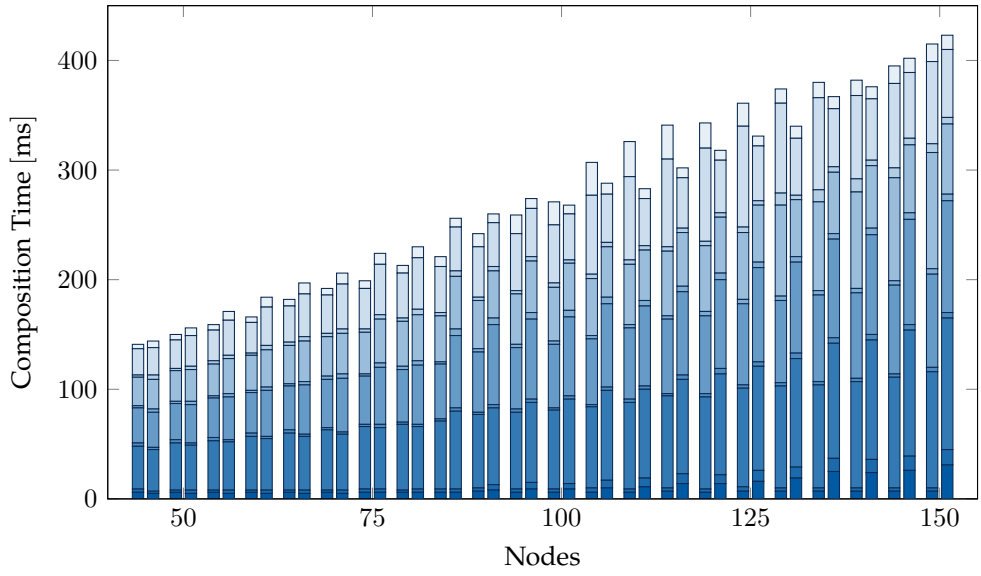


Figure A.4: Composition Time grouped by Composition Phases.
(left bars represent Simple-Sample, right bars represent Five-Roles)

into its ten individual phases, *i.e.*, from bottom to top: initialize (planning), send invites, wait for replies, send instantiates, wait again, send binding instructions, wait again,

send role activation, wait for successful response, and finally send collaboration activation. Especially for the last two phases of the composition process, the Simple-Sample requires much more time for the aforementioned system sizes. One reason for this is that messages on the PCC are processed in a single, synchronized thread. While in the Five-Roles scenario more roles have to be instantiated, the utilization of multiple threads is better than in the Simple-Sample scenario as just one role needs to be instantiated per virtualized node. Since all nodes are placed on one physical machine, this issue becomes a bottleneck.

Appendix B Code Listings

B.1 Concept

Listing B.1: Sample Implementation of the `LectureContext`.

```
1  package org.rosi.roledisco.samples.classroom;
2
3  /* import statements */
4
5  public class LectureContext implements Serializable {
6    private final String lectureName;
7    private final String description;
8    private final UUID lectureId;
9
10   public LectureContext(String lectureName, String description, UUID lectureId) {
11     this.lectureName = lectureName;
12     this.description = description;
13     this.lectureId = lectureId;
14   }
15
16   @Override
17   public int hashCode() {
18     return lectureId.hashCode();
19   }
20
21   @Override
22   public boolean equals(Object o) {
23     if (this == o) return true;
24     if (o == null || getClass() != o.getClass()) return false;
25     LectureContext that = (LectureContext) o;
26     return lectureId.equals(that.lectureId);
27   }
28 }
```

B.2 Implementation

Listing B.2: Xtext Grammar for the Role-based Collaboration Specification.

```
grammar de.tudresden.inf.rn.rosi.roledisco.CollaborationSpecification with org.
    eclipse.xtext.xbase.Xbase

generate collaborationSpecification "http://rn.inf.tu-dresden.de/rosi/roledisco/
    CollaborationSpecification"

import "http://www.eclipse.org/xtext/xbase/Xbase" as xBase
import "http://www.eclipse.org/xtext/common/JavaVMTypes" as types
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

CollSpec:
  importSection=XImportSection?
  module = Module;

Module:
  'module' name=QualifiedName
  collaboration = Collaboration;

Collaboration:
  'collaboration' name=ValidID '{'
  (
      context+=Context*
      (roles+=CoordinatorRole) &
      (roles+=NonCoordinatorRole+) &
      (constraints=RoleConstraints)? &
      (multiplicities=Multiplicities)?
  )
  '}';

Context:
  'context' featureType=JvmTypeReference featureName=ValidID;

CoordinatorRole:
  'coordinator' 'role' Role;

NonCoordinatorRole:
  'role' Role;

fragment Role:
  name=ValidID '{'
  context+=Context*
  features+=Feature*
  '}';

RoleConstraints:
  'constraints' '{'
    constraints+=RoleConstraint*
  '}';

RoleConstraint:
```

```
49   RoleProhibition | RoleImplication | RoleEquivalence;
50
51 RoleProhibition:
52   roleA=[Role]
53   '>-<'
54   roleB=[Role];
55
56 RoleImplication:
57   roleA=[Role]
58   '-->'
59   roleB=[Role];
60
61 RoleEquivalence:
62   roleA=[Role]
63   '<->'
64   roleB=[Role];
65
66 Multiplicities:
67   'multiplicities' '{'
68   multiplicities+=Multiplicity*
69   '}';
70
71 Multiplicity:
72   OneToOne | OneToMany;
73
74 OneToOne:
75   roleA=[Role]
76   ('one-to-one'|'to')
77   roleB=[Role];
78
79 OneToMany:
80   roleA=[Role]
81   (
82       toInfinity?='one-to-many' |
83       toInfinity?='to-many' |
84       'to' '(' howMany=INT ')'
85   )
86   roleB=[Role];
87
88 Feature:
89   Property | Operation;
90
91 Property:
92   type=JvmTypeReference name=ValidID ('=' body=XLiteral)?;
93
94 Operation:
95   (isPlayerQualifierSet?='player')? 'op' name=ValidID
96   '('(params+=FullJvmFormalParameter
97   (',' params+=FullJvmFormalParameter)*)?')'
98   (=>':' type=JvmTypeReference)?
99   body=XBlockExpression?;
100
101 PlayerFeatureCall returns xBase::XExpression:
102   PlayerLiteral
```

```
103   (=>({PlayerAssignment.assignable=current} ('.'|explicitStatic?="::") feature=[types
          ::JvmIdentifiableElement|FeatureCallID] OpSingleAssign) value=XAssignment
104   |=>({PlayerFeatureCall.memberCallTarget=current} ("."|nullSafe?="?."|explicitStatic
          ?="::"))
105   (explicitReturnType?='[' returnType=JvmTypeReference']')?
106   ('<' typeArguments+=JvmArgumentTypeReference (',' typeArguments+=
          JvmArgumentTypeReference)* '>')?
107   feature=[types::JvmIdentifiableElement|IdOrSuper] (
108       =>explicitOperationCall?='('
109       (
110           memberCallArguments+=XShortClosure
111           | memberCallArguments+=XExpression (',' memberCallArguments+=XExpression)*
112       )?
113       ')')?
114   memberCallArguments+=XClosure?
115   );
116
117 XPrimaryExpression returns xBase::XExpression:
118   XConstructorCall |
119   XBlockExpression |
120   XSwitchExpression |
121   XSynchronizedExpression |
122   XFeatureCall |
123   XLiteral |
124   XIfExpression |
125   XForLoopExpression |
126   XBasicForLoopExpression |
127   XWhileExpression |
128   XDoWhileExpression |
129   XThrowExpression |
130   XReturnExpression |
131   XTryCatchFinallyExpression |
132   XParenthesizedExpression |
133   PlayerFeatureCall;
134
135 XLiteral returns xBase::XExpression:
136   XCollectionLiteral |
137   XClosure |
138   XBooleanLiteral |
139   XNumberLiteral |
140   XNullLiteral |
141   XStringLiteral |
142   XTypeLiteral;
143
144 PlayerLiteral:
145   {PlayerLiteral} 'player';
```

Listing B.3: Deriving the Player Interface of a Role.

```
1 def void inferPlayerInterface(Role role, IJvmDeclaredTypeAcceptor acceptor, boolean
      isPreIndexingPhase, CollSpec collSpec) {
2   acceptor.accept(role.toInterface('''«collSpec.module.fullyQualifiedName».I«role.
        name»RolePlayer''',[])) [
3     setInterface = true
```

```
4     for(feature : role.features) {
5       if(feature instanceof Operation) {
6         if(feature.isPlayerQualifierSet) {
7           members += feature.toMethod(feature.name, feature.type) [
8             setDefault = false
9             setAbstract = true
10            documentation = feature.documentation
11            for(p : feature.params) {
12              parameters += p.toParameter(p.name, p.parameterType)
13            }
14          ]
15        }
16        if(null!==feature?.body)
17          for(PlayerFeatureCall c : EcoreUtil2.getAllContentsOfType(feature?.body,
                PlayerFeatureCall)) {
18            val node2 = NodeModelUtils.findNodesForFeature(c,XbasePackage.Literals.
                XABSTRACT_FEATURE_CALL__FEATURE).head
19            val name = node2.text.trim
20            members += role.toMethod(name,if(c.explicitReturnType ) c.returnType.type
                .typeRef else Void.TYPE.typeRef) [
21              setDefault = false
22              setAbstract = true
23            ]
24          }
25      }
26    }
27  ]
28 }
```

B.3 Evaluation

Listing B.4: Session Class used as Context Property.

```
1 package org.rosi.roledisco.samples.ds;
2
3 public class Session {
4   private final String sessionId;
5
6   public Session(String sessionId) {
7     this.sessionId = sessionId;
8   }
9
10  @Override
11  public boolean equals(Object o) {
12    if (this == o) return true;
13    if (o == null || getClass() != o.getClass()) return false;
14
15    Session session = (Session) o;
16
17    return getSessionId().equals(session.getSessionId());
18  }
```

```
19
20   @Override
21   public int hashCode() {
22     return getSessionId().hashCode();
23   }
24
25   public String getSessionId() {
26     return sessionId;
27   }
28 }
```

Listing B.5: Generated Collaboration Class of the Distributed Slideshow Scenario.

```
1  package org.rosi.roledisco.samples.ds;
2
3  import de.tudresden.inf.rn.rosi.roledisco.middleware.composition.coordination.
       CompositionManagement;
4  import de.tudresden.inf.rn.rosi.roledisco.model.AbstractCollaboration;
5  import de.tudresden.inf.rn.rosi.roledisco.model.annotations.Collaboration;
6  import de.tudresden.inf.rn.rosi.roledisco.model.annotations.Constraint;
7  import de.tudresden.inf.rn.rosi.roledisco.model.annotations.Context;
8  import de.tudresden.inf.rn.rosi.roledisco.model.annotations.Multiplicity;
9  import de.tudresden.inf.rn.rosi.roledisco.model.constraints.RoleConstraint;
10 import de.tudresden.inf.rn.rosi.roledisco.model.multiplicities.RoleLink;
11
12 @Collaboration(
13     coordinator = PresenterRole.class,
14     roles = ViewerRole.class,
15     kind = Collaboration.Kind.ONE_TO_MANY
16 )
17 @Constraint(
18     from = PresenterRole.class,
19     to = ViewerRole.class,
20     value = RoleConstraint.ROLE_PROHIBITION
21 )
22 @Multiplicity(
23     from = PresenterRole.class,
24     to = ViewerRole.class,
25     value = RoleLink.ONE_TO_MANY
26 )
27 public class DistributedSlideshowCollaboration extends AbstractCollaboration {
28
29   @Context
30   Session session;
31
32   public DistributedSlideshowCollaboration(Session session) {
33     super();
34     this.session = session;
35     CompositionManagement.compose(this);
36   }
37
38   public DistributedSlideshowCollaboration(PresenterRole coordinator) {
39     super(coordinator);
40     CompositionManagement.compose(this, coordinator);
```

```
41   }
42
43   public DistributedSlideshowCollaboration(PresenterRole coordinator, Session session
         ) {
44     super(coordinator);
45     this.session = session;
46     CompositionManagement.compose(this, coordinator);
47   }
48
49   public Session getSession() {
50     return session;
51   }
52
53   public void setSession(Session session) {
54     this.session = session;
55   }
56 }
```

Listing B.6: Generated Code of the Presenter Role.

```
1  package org.rosi.roledisco.samples.ds;
2
3  import de.tudresden.inf.rn.rosi.roledisco.middleware.dispatcher.Dispatcher;
4  import de.tudresden.inf.rn.rosi.roledisco.model.AbstractCoordinatorRole;
5
6  public class PresenterRole extends AbstractCoordinatorRole {
7
8    public void sendMessage(String msg) {
9      Dispatcher.getDispatcher().dispatchToRoles(this, ViewerRole.class,"receiveMessage
           ",msg);
10   }
11
12   public void receiveMessage(String msg){
13     Dispatcher.getDispatcher().dispatchToPlayer(this,"receiveMessage",msg);
14   }
15 }
```

Listing B.7: Complete Implementation of `DistributedSlideshow` Class.

```
1  package org.rosi.roledisco.samples.ds.ui;
2
3  /* import statements */
4
5  public class DistributedSlideshow {
6    private final JFrame root;
7    private List<Picture> pictures;
8    private int imageIndex = 0;
9    private JButton previousButton;
10   private JButton nextButton;
11   private JTextField feedback;
12   private JLabel image;
13   private JLabel session;
14   private JTextArea comments;
```

```
15   private JPanel rootPanel;
16   private JScrollPane scp1;
17   private ViewerRole viewerRole;
18   private PresenterRole presenterRole;
19
20   /**
21    * Player for the Presenter Role
22    */
23   public DistributedSlideshow(String sessionName, List<Picture> pictures) {
24     PresenterPlayerProvider.getInstance().addPresenter(sessionName, this);
25     new DistributedSlideshowCollaboration(new Session(sessionName));
26     this.pictures = pictures;
27     session.setText(sessionName);
28
29     root = new JFrame("Distributed␣Slideshow␣(" + sessionName + ")␣[Presenter]");
30     rootPanel.remove(feedback);
31     root.setContentPane(rootPanel);
32     root.pack();
33     root.setVisible(true);
34     root.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
35
36     image.setIcon(new ImageIcon(pictures.get(0).getAbsolutePath()));
37
38     nextButton.addActionListener(e -> {
39       int newImageIndex = (imageIndex + 1) % pictures.size();
40       if (presenterRole != null) {
41         presenterRole.setPicture(pictures.get(newImageIndex));
42       }
43       DistributedSlideshow.this.setPicture(pictures.get(newImageIndex));
44       imageIndex = newImageIndex;
45     });
46
47     previousButton.addActionListener(e -> {
48       int newImageIndex = (imageIndex - 1) % pictures.size();
49       if (newImageIndex < 0) imageIndex += pictures.size();
50       if (presenterRole != null) {
51         presenterRole.setPicture(pictures.get(newImageIndex));
52       }
53       DistributedSlideshow.this.setPicture(pictures.get(newImageIndex));
54       imageIndex = newImageIndex;
55     });
56   }
57
58   /**
59    * Player for the Viewer Role
60    */
61   public DistributedSlideshow(String sessionName) {
62     session.setText(sessionName);
63     rootPanel.remove(scp1);
64     rootPanel.remove(nextButton);
65     rootPanel.remove(previousButton);
66     feedback.addActionListener(e -> {
67       if (viewerRole != null) {
68         viewerRole.sendFeedback(feedback.getText());
69         feedback.setText("");
```

```
70        }
71      });
72      root = new JFrame("Distributed␣Slideshow␣(" + sessionName + ")␣[Guest]");
73      root.setContentPane(rootPanel);
74      root.pack();
75      root.setVisible(true);
76      root.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
77    }
78
79    private void receiveFeedback(String message) {
80      comments.append(String.format("%s\n", message));
81    }
82
83    private void setPicture(Picture picture) {
84      image.setIcon(new ImageIcon(picture.getAbsolutePath()));
85    }
86
87    /* UI Designer generated code */
88  }
```

Listing B.8: Context & Player Provider for the Presenter Role.

```
1  package org.rosi.roledisco.samples.ds;
2
3  /* import statements */
4
5  public class PresenterPlayerProvider implements PlayerProvider, ContextProvider {
6    private static final PresenterPlayerProvider myInstance = new
         PresenterPlayerProvider();
7
8    static {
9      Dispatcher.getDispatcher().addPlayerProvider(PresenterRole.class, getInstance());
10     ContextManager.getContextManager().addContextProvider(PresenterRole.class,
           getInstance());
11   }
12
13   private final Map<String, DistributedSlideshow> players = Maps.newHashMap();
14
15   public static synchronized final PresenterPlayerProvider getInstance() {
16     return myInstance;
17   }
18
19   @Override
20   public Collection<Context> getContext(Class<? extends AbstractRole> role) {
21     return Collections.emptyList(); // Presenter is not required to provide contexts
22   }
23
24   @Override
25   public boolean hasPlayer(Class<? extends AbstractRole> roleType, Context context) {
26     return true; // irrelevant as Presenter is manually triggered
27   }
28
29   @Override
30   public Object getPlayer(Class<? extends AbstractRole> roleType, Context context) {
```

```
31      Session session = (Session) context.getValueForKey(new ContextFeature("session"))
            .getValue();
32      return players.get(session.getSessionId());
33    }
34
35    public void addPresenter(String session, DistributedSlideshow slideshow) {
36      players.put(session, slideshow);
37    }
38  }
```

Listing B.9: Distributed Slideshow's Main Application.

```
1  package org.rosi.roledisco.samples.ds.ui;
2
3  /* import statements */
4
5  public class SlideshowMain extends JDialog {
6    private String session;
7    private JPanel contentPane;
8    private JButton buttonOK;
9    private JButton buttonCancel;
10
11   public SlideshowMain() {
12     setContentPane(contentPane);
13     setModalityType(ModalityType.MODELESS);
14     getRootPane().setDefaultButton(buttonOK);
15     buttonOK.addActionListener(e -> onOK());
16     buttonCancel.addActionListener(e -> onCancel());
17     setDefaultCloseOperation(DO_NOTHING_ON_CLOSE);
18     addWindowListener(new WindowAdapter() {
19       public void windowClosing(WindowEvent e) {
20         onCancel();
21       }
22     });
23     contentPane.registerKeyboardAction(e -> onCancel(), KeyStroke.getKeyStroke(
           KeyEvent.VK_ESCAPE, 0), JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);
24     session = null;
25     while (session == null || session.isEmpty())
26       session = JOptionPane.showInputDialog(this, "Session␣Name?", "Session␣Name",
             JOptionPane.QUESTION_MESSAGE);
27     ViewerPlayerProvider.getInstance().addSession(session);
28   }
29
30   public static void main(String[] args) {
31     SlideshowMain dialog = new SlideshowMain();
32     dialog.pack();
33     dialog.setVisible(true);
34   }
35
36   private void onOK() {
37     JFileChooser fileChooser = new JFileChooser();
38     fileChooser.addChoosableFileFilter(new FileFilter() {
39       @Override
40       public boolean accept(File f) {
```

```
41          if (f.isDirectory()) {
42            return true;
43          }
44          String extension = FileUtils.getExtension(f);
45          if (extension != null) {
46            return extension.equals(FileUtils.bmp) ||
47                extension.equals(FileUtils.jpeg) ||
48                extension.equals(FileUtils.jpg) ||
49                extension.equals(FileUtils.png);
50          }
51          return false;
52        }
53
54        @Override
55        public String getDescription() {
56          return "Images␣(*.bmp,␣*.jpeg,␣*.jpg,␣*.png)";
57        }
58      });
59      fileChooser.setMultiSelectionEnabled(true);
60      if (fileChooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION) {
61        File[] files = fileChooser.getSelectedFiles();
62        if (files.length > 0) {
63          new DistributedSlideshow(session, Lists.newArrayList(files).stream().map(file
                -> new Picture(file.getAbsolutePath())).collect(Collectors.toList()));
64          dispose();
65        }
66      }
67    }
68
69    private void onCancel() {
70      dispose();
71      System.exit(0);
72    }
73
74    /* UI Designer generated code */
75  }
```

Listing B.10: Discovery Service's Implementation.

```
1  package de.tudresden.inf.rn.rosi.roledisco.middleware.discovery;
2
3  /* import statements */
4
5  @Singleton
6  public class DiscoveryService {
7    private final EventBus eventBus;
8    private DirectoryService directoryService;
9    private ScheduledExecutorService timer;
10   private InfrastructureAbstractionLayer infrastructure;
11   private ContextManager contextManager;
12   private Dispatcher dispatcher;
13   private MessageQueue messageQueue;
14   private CollaborationManager collaborationManager;
15
```

```
16  @Inject
17  public DiscoveryService(MessageQueue messageQueue, DirectoryService
        directoryService, InfrastructureAbstractionLayer infrastructure, ContextManager
         contextManager, Dispatcher dispatcher, CollaborationManager
        collaborationManager, EventBus eventBus) {
18    this.directoryService = directoryService;
19    this.infrastructure = infrastructure;
20    this.messageQueue = messageQueue;
21    this.contextManager = contextManager;
22    this.dispatcher = dispatcher;
23    this.collaborationManager = collaborationManager;
24    this.eventBus = eventBus;
25    this.timer = Executors.newScheduledThreadPool(5);
26
27    eventBus.register(this);
28    messageQueue.register(this);
29    collaborationManager.init();
30    timer.scheduleAtFixedRate(this::heartbeat, 0, 1000, TimeUnit.MILLISECONDS);
31  }
32
33  @Subscribe
34  private void newCollaborationSpecification(NewCollaborationSpecificationEvent e) {
35    publish(Collections.unmodifiableCollection(collaborationManager.
          getCollaborationSpecifications()), null);
36  }
37
38  @Subscribe
39  private void newSubsystemDetected(NewSubsystemEvent event) {
40    if (event.getDiscoveryType() == NewSubsystemEvent.DiscoveryType.BY_BROADCAST)
41      publish(Collections.unmodifiableCollection(collaborationManager.
            getCollaborationSpecifications()), event.getSubsystem());
42  }
43
44  @Subscribe
45  private void contextUpdate(ContextUpdate contextUpdate) {
46    publish(Lists.newArrayList(contextUpdate.getSpecification()), null);
47  }
48
49  private Collection<RoleAnnouncementMessage> createMessages(
        CollaborationSpecification specification) {
50    Collection<RoleAnnouncementMessage> messages = new HashSet<>();
51    if (isDiscoverable(specification.getTheCoordinatorRole())) {
52      messages.add(createMessage(specification.getTheCoordinatorRole(), specification
            .getTheCollaboration()));
53    }
54    for (Class<? extends AbstractRole> roleType : specification.getOtherRoles()) {
55      if (isDiscoverable(roleType)) {
56        messages.add(createMessage(roleType, specification.getTheCollaboration()));
57      }
58    }
59    return messages;
60  }
61
62  private RoleAnnouncementMessage createMessage(Class<? extends AbstractRole>
        roleType, Class<? extends AbstractCollaboration> theCollaboration) {
```

```
63      return RoleAnnouncementMessage.create(theCollaboration, roleType, contextManager.
            getContext(roleType));
64    }
65
66    private boolean isDiscoverable(Class<? extends AbstractRole> roleType) {
67      Collection<Context> contexts = contextManager.getContext(roleType);
68      if (contexts.isEmpty()) {
69        if (roleType.getSuperclass().equals(AbstractCoordinatorRole.class))
70          return dispatcher.hasPlayer((Class<? extends AbstractCoordinatorRole>)
                roleType);
71        else
72          return dispatcher.hasPlayer((Class<? extends AbstractNonCoordinatorRole>)
                roleType, null);
73      } else {
74        for (Context context : contexts) {
75          if (isDiscoverable(roleType, context)) return true;
76        }
77      }
78      return false;
79    }
80
81    private boolean isDiscoverable(Class<? extends AbstractRole> roleType, Context
          context) {
82      return dispatcher.hasPlayer((Class<? extends AbstractNonCoordinatorRole>)
            roleType, context);
83    }
84
85    private void publish(Collection<CollaborationSpecification> specifications,
          @Nullable AdaptiveSubsystem receiver) {
86      Collection<RoleAnnouncementMessage> messages = new HashSet<>();
87      for (CollaborationSpecification specification : specifications) {
88        messages.addAll(createMessages(specification));
89      }
90      if (receiver == null) {
91        for (RoleAnnouncementMessage message : messages) {
92          infrastructure.publish(message);
93        }
94      } else {
95        for (RoleAnnouncementMessage message : messages) {
96          infrastructure.send(receiver, message);
97        }
98      }
99    }
100
101   @Subscribe
102   @AllowConcurrentEvents
103   private void processMessage(RoleAnnouncementMessage msg) {
104     directoryService.addDiscoveryInformation(msg.getSource(), msg.getCollaboration(),
            msg.getRole(), msg.getContext());
105     if (directoryService.reportSubsystem(msg.getSource())) {
106       eventBus.post(new NewSubsystemEvent(msg.getSource(), msg.isBroadcast()));
107     }
108   }
109
110   private void terminate() {
```

```
111     timer.shutdown();
112     messageQueue.unregister(this);
113   }
114
115   private void heartbeat() {
116     infrastructure.publish(new HeartbeatMessage());
117   }
118
119   @Subscribe
120   @AllowConcurrentEvents
121   private void processMessage(HeartbeatMessage msg) {
122     if (directoryService.reportSubsystem(msg.getSource())) {
123       eventBus.post(new NewSubsystemEvent(msg.getSource(), msg.isBroadcast()));
124     }
125   }
126 }
```

Listing B.11: Implementation of `checkForStateCompletionAndContinue`.

```
1  private synchronized void checkForStateCompletionAndContinue(CompositionState
       expectedState, Collection<? extends Message> requests, Collection<? extends
       Message> responses, Collection<? extends Message> errorResponses, final Timer
       timerToBeCancelled) {
2    if (expectedState.equals(states.peek()) && responses.size() + errorResponses.size()
         == requests.size()) {
3      timerToBeCancelled.cancel();
4      timerToBeCancelled.purge();
5      if (errorResponses.size() > 0)
6        setState(expectedState.failure());
7      else {
8        setState(expectedState.success());
9        requests.clear();
10       responses.clear();
11       errorResponses.clear();
12     }
13     this.compose();
14   }
15 }
```